VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCA

2010 **Bc. Radovan Baranec**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Posúdenie vhodnosti virtuálnych strojov na dekódovanie paketov

Consideration of Virtual Machines Suitability for Packet Decoding

2010 **Bc. Radovan Baranec**

Zadanie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne, uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 24. apríla 2010 .....

Chcel by som poďakovať pánovi Petrovi Kopeckému za jeho čas strávený pri konzultáciách a poskytnuté rady. Ďalej by som chcel poďakovať Ing. Petrovi Olivkovi za odborné vedenie počas tvorby diplomovej práce a podnetné pripomienky k textu.

Abstrakt

Táto diplomová práca posudzuje vhodnosť aplikačných virtuálnych strojov LLVM a Parrot na dekódovanie resp. filtrovanie sieťovej prevádzky a to nie len z pohľadu výkonu ale aj z pohľadu zložitosti realizácie. Porovnáva implementáciu jednoúčelového sieťového filtra medzi virtuálnymi strojmi navzájom a s riešeniami založenými na kompilovaných a skriptovacích jazykoch. Výsledkom práce je odporúčanie virtuálneho stroja LLVM.

Kľúčové slová

Virtuálny stroj, dekódovanie, filtrovanie, paket, rámec, Parrot, LLVM, kompilátor.

Abstract

This diploma thesis considers suitability of application virtual machines LLVM and Parrot for network traffic decoding resp. filtering not only from performance point of view but also difficulty of realization. It compares implementation of single-purpose network filter between virtual machines themselves and solutions based on compiled and scripting languages. The result is recommendation of virtual machine LLVM.

Keywords

Virtual machine, decoding, filtering, packet, frame, Parrot, LLVM, compiler.

Zoznam použitých symbolov a skratiek

API	- Application Programming Interface
BPF	- Berkeley Packet Filter
BSD	- Berkeley Software Distribution
CLR	- Common Language Runtime
CPU	- Central Processing Unit
CRC	- Cyclic Redundancy Check
GC	- Garbage Collector
IP	- Internet Protocol
IR	- Intermediate Representation
IT	- Information Technology
JIT	- Just in Time
LLVM	- Low Level Virtual Machine
NCI	- Native Call Interface
NetPDL	- Network Protocol Description Language
NetPE	- Network Processing Element
NetPFL	- Network Packet Filtering Language
NetVM	- Network Virtual Machine
OSI	- Open System Interconnection
PASM	- Parrot Assembly
PAST	- Parrot Abstract Syntax Tree
PBC	- Parrot Byte Code
PCAP	- Packet Capture
PES	- Parrot Embedding System
PIR	- Parrot Intermediate Representation
PMC	- Polymorphic Container
RISC	- Reduced Instruction Set Computer
SSA	- Static Single Assignment
TCP	- Transmission Control Protocol
VM	- Virtual Machine

Obsah

Prílohy

Zoznam obrázkov

Zoznam výpisov zdrojového kódu

Úvod

Počítačové siete sú neoddeliteľnou súčasťou nášho každodenného života. Stretávame sa s nimi v práci, doma, na úradoch, ... najčastejšie napríklad v podobe Internetu. Stali sa neodmysliteľnou súčasťou veľkého počtu firiem, ktoré by bez nich nemohli fungovať. Pre mnohé z nich je správny chod siete natoľko kritický, že venujú nemalé úsilie návrhu architektúry a samotnej správe siete. Niektoré úkony spojené práve so správou siete ukladá aj samotný zákon[1] ČR, ktorý prikazuje prevádzkovateľom verejných telekomunikačných sietí uchovávať niekoľko mesiacov údaje o elektronickej komunikácii. Tieto činnosti bývajú často podporené aplikáciami na analýzu siete, ktoré umožňujú identifikovať úzke miesta v sieti, uchovávať sieťovú prevádzku len konkrétnych protokolov, atď.

Práve v tejto oblasti sa angažuje zadávateľ diplomovej práce, firma LinuxBox.cz[2], ktorá sa zaoberá poskytovaním komplexných riešení a podpory v oblasti informačných technológií. Okrem iného vyvíja aj produkt FoxStat[3] s modulom NetStat, ktorý analyzuje sieťovú prevádzku. V súčasnosti je jadro modulu NetStat implementované v jazyku C. Toto riešenie so sebou prináša obrovskú výhodu v podobe výkonu, avšak na druhej strane nevýhodu v podobe veľmi zložitej realizácie. Pri zmenách sieťového protokolu, filtra a mnohých iných je potrebné upraviť zdrojový kód, všetko opäť preložiť a spustiť. Riešeniam založeným na kompilovaných jazykoch chýbajú vlastnosti ako dynamika (zmena kódu za behu), jednoduchosť (automatická správa pamäte), platformová nezávislosť, ktoré so sebou prinášajú virtuálne stroje. Ďalšou prednosťou virtuálnych strojov je možnosť implementácie jednoduchých jazykov pre zadávanie filtrov podobnému napríklad jazyku BPF alebo pre popis protokolov, na základe ktorého by sa generoval dekodér. Medzi nevýhody patrí predovšetkým výkon, avšak s použitím rôznych technológií ako napríklad JIT kompilácia sa stáva tento deficit zanedbateľný. Práve o použití virtuálnych strojov sa uvažuje pri návrhu novej architektúry jadra modulu NetStat.

Cieľom diplomovej práce je posúdiť vhodnosť aplikačných virtuálnych strojov Parrot a LLVM na dekódovanie a filtrovanie sieťovej prevádzky. V assemblery virtuálnych strojov Parrot a LLVM implementujeme pod operačným systémom typu Linux jednoúčelové filtre. Tie porovnáme z pohľadu zložitosti realizácie a z pohľadu výkonu medzi virtuálnymi strojmi navzájom a s riešeniami založenými na jazykoch C a Python (využívajúceho Scapy), BPF implementovaného v knižnici PCAP a knižnici NetBee. Súčasťou práce je aj implementácia kompilátora z minijazyka inšpirovaného BPF do assembleru LLVM a realizácia filtra založeného na tomto minijazyku. Úvodné dve kapitoly vysvetľujú samotný názov diplomovej práce. V prvej sa dočí-

[1] Vyhláška č. 485/2005 Zb. Ministerstva Informatiky Českej republiky zo dňa 7.12.2005 o rozsahu prevádzkových a lokalizačných údajov, dobe ich uchovávania a forme a spôsobe ich predania orgánom oprávneným k ich využívaniu.
[2] http://www.linuxbox.cz/
[3] http://www.foxstat.com/

tame, čo sú to vlastne aplikačné virtuálne stroje. Druhá kapitola vysvetľuje pojem dekódovanie paketov a zároveň poskytuje návod ako implementovať filter v jazyku C. Kapitoly nasledujúce za ňou popisujú jednotlivé riešenia a realizáciu filtrov v každom z nich resp. použité aplikácie (tcpdump, nbeedump). Záver diplomovej práce pojednáva o vykonaných testoch, plusoch a mínusoch jednotlivých riešení.

1 Aplikačný virtuálny stroj

Aplikačný virtuálny stroj (1) je počítačový softvér, ktorý beží v operačnom systéme ako normálna aplikácia. Jeho cieľom je poskytnúť platformovo nezávislé programovacie a behové prostredie, ktoré zakrýva detaily hardvéru a operačného systému. Umožňuje tak vykonávať programy na rôznych platformách rovnakým spôsobom (strojovo alebo platformovo nezávislé[4] programy). Každý virtuálny stroj má svoj vlastný assembler alebo podporuje nejaký jazyk vyššej úrovne pre zápis aplikácií. Najznámejší príklad aplikačného virtúalneho stroja je Java virtuálny stroj pre programovací jazyk Java. V rámci diplomovej práce sa zoznámime s virtuálnymi strojmi[5] LLVM, Parrot a úzko špecializovaným NetBee.

Virtuálne stroje delíme z pohľadu architektúry (podľa toho akú formu má pamäť) na zásobníkové a registrové. Obecne zásobníkové VM vyžadujú viac inštrukcií na vykonanie niektorých operácií, naproti tomu registrové VM kvôli potrebe adresovať zdrojový a cieľový register majú dlhšie inštrukcie. Nedá sa však všeobecne povedať, ktorý typ je efektívnejší.

Programy realizované vo virtuálnych strojoch sú vykonávané pomocou interpretera. Interpreter je program, ktorý umožňuje vykonávať (interpretovať) zápis iného programu v danom programovacom jazyku (interpretovaný jazyk). Na rozdiel oproti prekladačom nie je nutné kompilovať program do strojového kódu cieľového procesora, čo umožňuje ľahší prenos medzi rozdielnymi platformami. Mnohé interpretery resp. interpretované jazyky so sebou prinášajú aj ďalšie výhody ako rýchlosť vývoja aplikácie (na chyby v zdrojovom kóde často upozorňujú už počas editácie programu), jednoduchšia správa pamäte (podpora GC, premenné nie sú viazané na fyzické adresy v operačnej pamäti, je možné ich jednoduchšie premapovať a zabrániť tak fragmentácii), reflexia, dynamické typovanie, ... Medzi hlavné nevýhody interpreterov však patrí ich výkon, spravidla bývajú oveľa pomalšie ako kompilované jazyky. Je to spôsobené tým, že interpreter musí pri každom spustení programu vykonať analýzu kódu (run-time analýza) a potom uskutočniť požadovanú akciu.

Existujú rôzne kompromisy medzi rýchlosťou vývoja pri použití interpretovaných jazykov a rýchlosťou programu pri kompilovaných jazykoch. Niektoré jazyky umožňujú interpretovaný a kompilovaný kód volať navzájom a zdieľať premenné. To znamená, že po odladení a otestovaní celej alebo len časti aplikácie pod interpreterom je možné ju nakompilovať a využívať tak rýchlejšie vykonávanie. Mnohé interpretery nevykonávajú priamo zdrojový kód ale prevádzajú ho do optimalizovanej, kompaktnejšej formy (bytekódu).

Ďalšia často využívaná možnosť je JIT kompilácia. Jedná sa o technológiu, ktorá je založená na dvoch ideách: kompilácia do bytekódu a dynamická kompilácia. Vykonávaný program je za

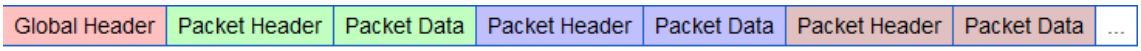
[4] Pozor, nejedná sa však o univerzálnu nezávislosť. Rôzne operačný systémy používajú rôznu adresárovú štruktúru, rôzne oddeľovače ciest, ... Tieto rozdiely musíme explicitne ošetriť sami.
[5] V ďalšom texte budeme rozumieť pod pojmom virtuálny stroj aplikačný virtuálny stroj.

behu prekladaný do strojového kódu, čo značne zvyšuje výkon. Je to vďaka uchovávaniu už preložených častí kódu, namiesto neustáleho vyhodnocovania každého riadku kódu ako je tomu v prípade interpretovania.

2 Dekódovanie paketov

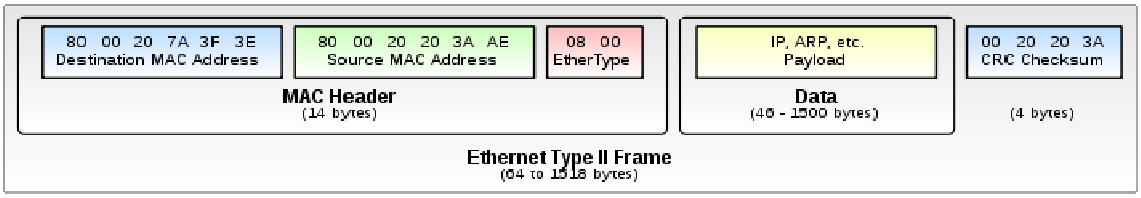
V tejto kapitole si vysvetlíme pojem dekódovanie paketov a podrobne popíšeme jednoúčelový filter, ktorý budeme postupne realizovať v jazyku C a v assemblery virtuálnych strojov LLVM a Parrot. Princíp filtrovania a dekódovania je v každom jazyku[6] rovnaký, preto bude detailne popísaný práve v jazyku C, ktorý je všeobecne známy.

Dáta z jedného počítača do druhého sú prenášané po sieti pomocou entít nazývaných pakety. Tie majú presne špecifikovanú štruktúru (model ISO/OSI, model TCP/IP), ktorá je závislá na použitej technológií a obmedzenú maximálnu dĺžku (zvyčajne 1500 bytov). Pri kopírovaní bežného textového súboru po sieti sa tak medzi počítačmi prenesú stovky až tisíce paketov.



Obrázok 1: Formát súboru knižnice PCAP

Asi najznámejšie aplikácie, ktoré potrebujú dekódovať pakety sú firewally. Úlohou týchto programov je zabrániť prenosu nechcených paketov (napríklad blokovanie stránok s pornografiou) pomocou presne špecifikovaných pravidiel. Na to, aby mohli aplikovať príslušné pravidlá, potrebujú poznať zloženie prenášaných paketov. Inak povedané, musia jednotlivé pakety dekódovať. To znamená, lokalizovať príslušné polia v štruktúre paketov, preložiť ich z binárnej formy do decimálnej prípadne hexadecimálnej a porovnať hodnoty týchto polí so špecifikovanými pravidlami. Na základe týchto porovnaní resp. aplikovaní filtračných pravidiel môžu jednotlivé pakety buď zablokovať alebo pustiť ďalej.



Obrázok 2: Hlavička Ethernet II rámca

2.1 Jednoúčelový filter v jazyku C

Jazyk C je statický imperatívny (procedurálny) kompilovaný jazyk, ktorého história sa píše od roku 1972. V súčasnosti spolu s jazykom C++ je skoro výhradne používaný v aplikáciách závislých na výkone (napr. real-time spracovanie sieťovej prevádzky). Jedná sa o jeden z najpoužívanejších jazykov vôbec, je v ňom naprogramovaný aj samotný operačný systém Linux. Vďaka svojej dlhej histórii sú prekladače pre tento jazyk skoro dokonale optimalizované a rozšírené na väčšinu platforiem. Neobsahuje automatickú správu pamäte v podobe GC, na

[6] V spojení s virtuálnymi strojmi chápeme jazyk ako assembler daného virtuálneho stroja.

druhú stranu umožňuje priamy prístup do operačnej pamäte (v kernel móde, ring0, ...), podporuje pointerovú aritmetiku, ktorá zjednodušuje a uľahčuje prácu s pamäťou, atď.

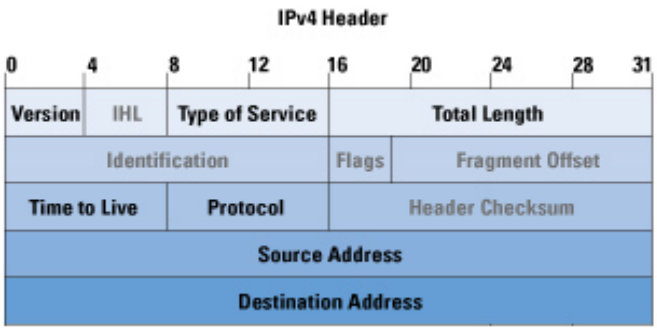
Slovo jednoúčelový je v názve podkapitoly použité zámerne, jedná sa o filter s dekodérom, ktorý nedekóduje celý paket ale iba dopredu dohodnuté protokoly podľa modelu OSI. Konkrétne sa jedná o protokoly IPv4->TCP, pričom filter je zadaný pevne a akceptuje len sieťovú prevádzku so zdrojovým alebo cieľovým portom 80 protokolu TCP. Výnimku tvorí LLVM, kde okrem jednoúčelového filtra implementujeme aj kompilátor pre minijazyk na zadávanie filtrov.

```
void decode(char* packet, unsigned int packet_len) {
      // zisti protokol zapuzdreny v naklade
      int type =  (packet[12] << 8) | packet[13];

      switch(type) {
            case ETHERNET_TYPE_IP:
                  decode_ip(packet, packet_len, ETHERNET_HEADER_LEN);
                  break;
            default:
                  break;
      }
}
```

Výpis 1: Dekódovanie hlavičky Ethernetu II v C

Ako jednotná testovacia úloha pre všetky implementované programy bola zvolená filtrácia sieťovej prevádzky z dopredu pripraveného súboru vo formáte (1) knižnice PCAP[7] (viď. obrázok 1). Aby sme neskôr pri porovnávaní výkonu jednotlivých riešení nenarazili na obmedzenie rýchlosti čítania z disku použili sme systémové volanie mmap. To umožňuje dopredu načítať celý súbor do operačnej pamäte, bez nutnosti pristupovať k pevnému disku počas samotného spracovania. Keďže sme zvolili práve takéto riešenie, nie je možné na čítanie paketov zo súboru použiť funkcie z knižnice PCAP priamo na to určené. Namiesto toho je do programu zahrnutý vlastný parser súborov vo formáte knižnice PCAP. Samotný program v jazyku C sa skladá z dvoch modulov *filter.c a decoder.c*, ktoré sú na priloženom CD v adresári *C/PacketFilter/src/*.



Obrázok 3: Hlavička IP datagramu

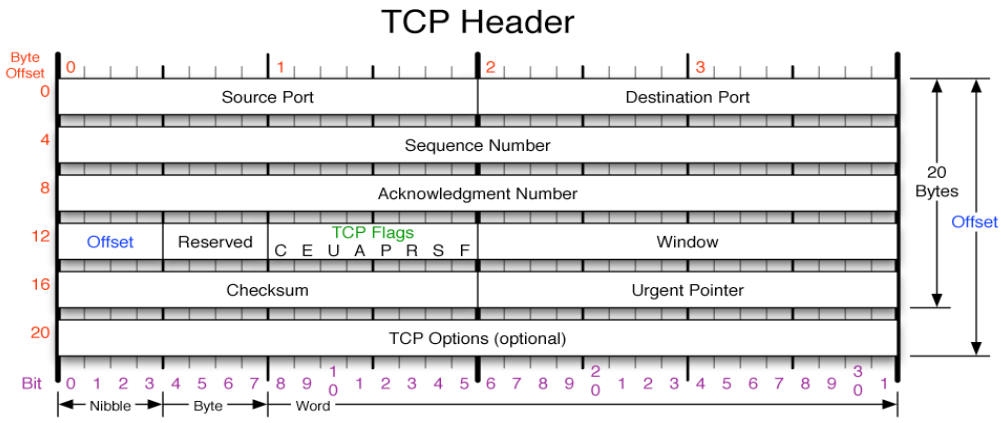
[7] http://www.tcpdump.org/

```
void decode_ip(char* packet, unsigned int packet_len,
               unsigned int offset) {
      //11110000 == 240
      char type = (packet[offset] & 240) >> 4;
      if (type == 4) {
            decode_ipv4(packet, packet_len, offset);
      }
      if (type == 6) {
            //decode_ipv6
      }
}
```

Výpis 2: Dekódovanie IP protokolu v C

2.1.1 Modul decoder.c

Úlohou tohto modulu ako napovedá samotný názov je dekódovanie paketu. Modulu resp. funkcii sú predané dáta paketu bez hlavičky (viď. obrázok 1). Pripomeňme si, že dáta nie sú reprezentované žiadnou dátovou štruktúrou (pozor, štruktúra paketu a reprezentácia dát paketu sú dve odlišné veci), jedná sa o súvislý úsek pamäte resp. pole, ktoré obsahuje položky typu char. Na to, aby sme zistili hodnotu zdrojového a cieľového portu protokolu TCP, musíme začať s dekódovaním na najnižších vrstvách, konkrétne na dátovej vrstve (Data Link Layer), ktorej súčasťou je štandard Ethernet II. Keďže máme presne vymedzené hodnoty ktoré nás zaujímajú, nie je nutné dekódovať celú hlavičku Ethernetu. Postačuje prečítať pole *Ether type* (viď obrázok 2), popisujúce, ktorý protokol je zapuzdrený v náklade resp. dátach rámca[8]. V prípade, že typ sa nezhoduje s IP, dekódovanie končí a rámec je odmietnutý. V opačnom prípade nasleduje dekódovanie protokolu IP na sieťovej vrstve. Výpis 1 zobrazuje dekódovanie hlavičky Ethernetu.



Obrázok 4: Hlavička TCP paketu

[8] Slová rámec, datagram a paket sú často laickou ale aj odbornou verejnosťou používané ako synonymá. V našom texte pokiaľ hovoríme o konkrétnych protokol na konkrétnych vrstvách modelu OSI, používame termíny Ethernet rámec, IP datagram, TCP paket. Pokiaľ hovoríme o obecnej entite prenášanej po sieti, hovoríme o pakete.

Z hlavičky IP datagramu (viď obrázok 3) musíme najskôr zistiť verziu protokolu IP z poľa *Version*, lebo v našom prípade nás zaujíma len IP verzie 4. Ak sa skutočne jedná o verziu 4, musíme ešte zistiť veľkosť hlavičky IP datagramu z poľa *IHL* a protokol z poľa *Protocol*, ktorý je zapuzdrený v náklade datagramu. Ak sa jedná o protokol TCP, pokračujeme ďalej v dekódovaní na transportnej vrstve. Výpis 2 zobrazuje dekódovanie protokolu IP.

Z hlavičky TCP paketu (viď obrázok 4) nám už stačí len prečítať polia *Source port* a *Destination port* a uložiť ich do globálnych premenných, ktoré neskôr využije modul *filter.c.* Výpis 3 zobrazuje ukážku dekódovania protokolu TCP.

```
void decode_tcp(char* packet, unsigned int packet_len,
                unsigned int offset) {
      // globalne premenne
      tcp_src_p = packet[offset] << 8 | packet[offset + 1];
      tcp_dst_p = packet[offset + 2] << 8 | packet[offset + 3];
}
```

Výpis 3: Dekódovanie protokolu TCP v C

2.1.2 Modul filter.c

Úlohou tohto modulu je namapovať filtrovaný súbor do operačnej pamäte, čítať jednotlivé pakety zo súboru pomocou vlastného parsera a posielať ich ďalej dekodéru. Nakoniec aplikujeme na dekódovaný paket filtračné pravidlo, v našom prípade zdrojový alebo cieľový port protokolu TCP sa musí rovnať 80. Výpis 4 zobrazuje ukážku časti zdrojového kódu resp. hlavného cyklu, ktorá získa paket zo súboru, pošle ho dekodéru a nakoniec aplikuje filtračné pravidlo.

```
while (1) {
      // precita paket
      packet_len = getPacket(&packet, file_address, file_size);
      if (packet_len == 0) {
            break;
      }
      // dekoduje paket
      decode(packet, packet_len);
      // aplikuje filter
      if (tcp_src_p == 80 || tcp_dst_p == 80) {
            acc_packets++;
      }
      else {
            rej_packets++;
      }
}
```

Výpis 4: Hlavný cyklus filtra v C

2.1.3 Preloženie a spustenie

V zložke *C/PacketFilter/bin/* je priložený *makefile*, ktorý spustíme príkazom *make.* Ten preloží program a vytvorí spustiteľný súbor *packetFilter*. Program je preložiteľný a spustiteľný ako na

32 bit tak aj na 64 bit systémoch. Cesta k testovaciemu súboru sa zadáva ako parameter príkazového riadku. Ukážka preloženia a spustenia:

make – preloží a nalinkuje zdrojové súbory
./packetFilter "cesta k súboru" - spustí filter

3 Jednoúčelový filter v jazyku Python

Jazyk Python bol zvolený ako zástupca dynamického imperatívneho jazyka, ktorý je interpretovaný. Obsahuje niekoľko programovacích paradigiem, objektovo orientované, funkcionálne, reflektívne paradigma. Za svoju dynamičnosť však platí výkonom, preto je najčastejšie používaný na skriptovanie alebo rýchle prototypovanie. Existuje niekoľko projektov, ktoré prinášajú JIT kompiláciu pre Python, ale jedná sa skôr o amatérske projekty bez oficiálnej podpory (Psyco, Unladen Swallow využívajúci LLVM JIT). Obsahuje automatickú správu pamäte v podobe GC, dynamické typovanie, avšak nepodporuje pointre a pointerovú aritmetiku, atď.

Filter v jazyku Python sa odlišuje od filtra popísaného v kapitole 1 tým, že nemá implementovaný vlastný dekodér ale používa projekt Scapy (2). Tento projekt obsahuje množstvo užitočných funkcií medzi ktoré patrí aj čítanie súborov vo formáte knižnice PCAP a dekódovanie sieťovej prevádzky. Vzniká otázka, prečo v jazyku Python nebol implementovaný filter za použitia technológii popísaných v kapitole 1. Od tohto jazyka sa už dopredu neočakával žiaden oslnivý výkon, ale skôr ukážka jednoduchosti implementácia filtra použitím dynamického jazyka. Samotný zdrojový kód zaberá 32 riadkov a nachádza sa v adresári *Python\PythonFilter\src* na priloženom CD. Výpis 5 zobrazuje ukážku skoro celého programu resp. hlavného cyklu.

```
for paket in scapy.utils.PcapReader(sys.argv[1]):
    tcp = paket.getlayer('TCP')
    if tcp:
        if tcp.sport == 80 or tcp.dport == 80:
            acc_packets += 1
        else:
            rej_packets += 1
    else:
        rej_packets += 1
```

Výpis 5: Hlavný cyklus filtra v Pythone

Už počas prvých testov funkčnosti sme však zistili veľmi slabý výkon filtra. Malý testovací súbor (27MB), ktorý filter v jazyku C spracoval okamžite (čas bol v podstate nemerateľný), spracovával filter v jazyku Python skoro 1 min. Na základe toho sme riešenie v jazyku Python úplne vylúčili z výkonnostných testov v závere diplomovej práce.

3.1 Preloženie a spustenie

Projekt resp. knižnica Scapy je bežne dostupná v operačných systémoch typu Linux. Na jej stiahnutie použite inštalačný manažér dostupný vo vašej distribúcii (yum Fedora, aptitude Ubuntu, ...). Keďže Python je interpretovaný jazyk, nie je potrebné žiadne preloženie. Môžeme však využiť preloženie zdrojového kódu do bytekódu. Ukážka preloženia a spustenia:

python –c pythonFilter.py - vznikne súbor s príponou .pyc
p*ython pythonFilter.pyc "cesta k súboru"* - spustenie filtra

4 Berkeley Packet Filter

BPF (3) poskytuje na systémoch založených na Unixe surové rozhranie (raw interface) na úrovni dátovej vrstvy, ktoré umožňuje zachytávať a posielať pakety. Ak sieťová karta podporuje promiskuitný mód, umožňuje BPF nastaviť kartu do tohto módu a zachytávať sieťovú prevádzku určenú pre iných hostiteľov.

Okrem iného podporuje BPF aj filtrovanie prevádzky, ktoré je implementované ako interpreter jazyka pre virtuálny stroj BPF. Programy v tomto jazyku sú schopné čítať dáta z paketu, vykonávať aritmetické operácia nad dátami, porovnávať výsledky voči konštantám, testovať bity vo výsledkoch a na základe zhodnotenia výsledkov prijať alebo odmietnuť paket. Na niektorých platformách existuje JIT kompilácia na konvertovanie inštrukcií virtuálneho stroja BPF do strojové jazyka danej platformy. Aj vďaka tomu sa niekedy na BPF odkazuje skôr ako na filtrovací mechanizmus než ako na celé rozhranie. BPF bolo vybrané do diplomovej práce zámerne, pretože v sebe skrýva princípy filtrovania prevádzky pomocou virtuálneho stroja. Ukážka jazyka BPF:

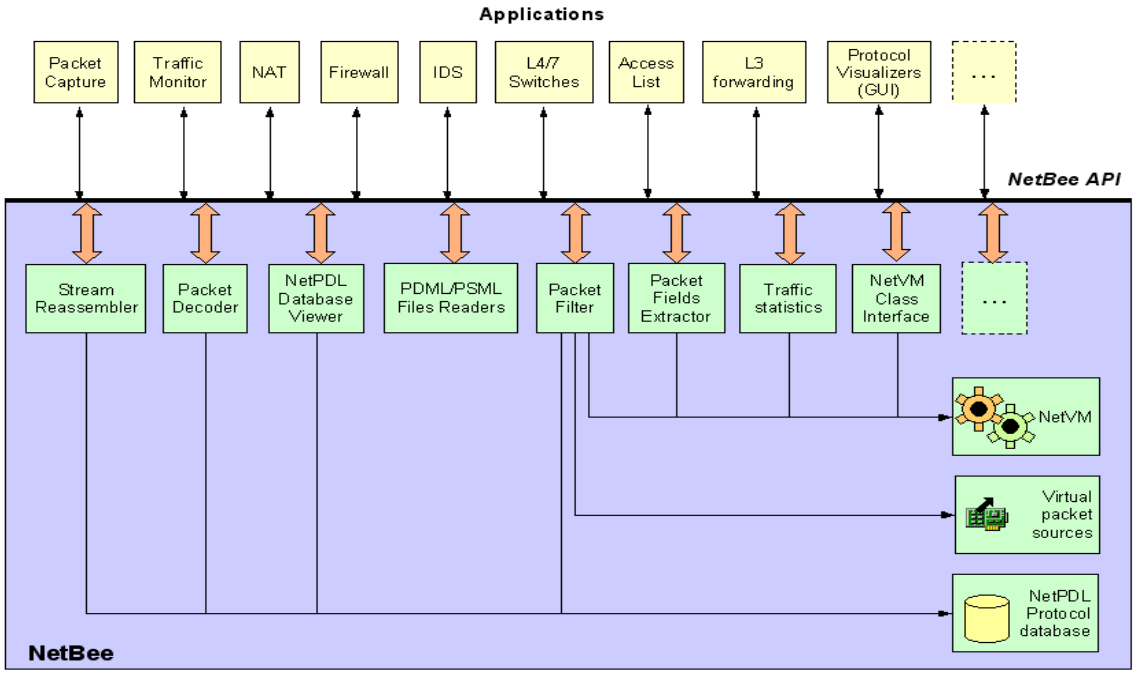
- ***dst port*** *num* – výraz je pravdivý ak paket je ip->tcp, ip->udp, ipv6->tcp, ipv6->udp a má cieľový port hodnoty num
- ***src port*** *num* - výraz je pravdivý ak paket je ip->tcp, ip->udp, ipv6->tcp, ipv6->udp a má zdrojový port hodnoty num

V užívateľskom móde poskytuje interpreter jazyka BPF knižnica PCAP. Jeden z nástrojov využívajúcich túto knižnicu je tcpdump (4), ktorý je zahrnutý aj v porovnávaní v závere práce. Tento nástroj je bežne dostupný v operačných systémoch Linux. V podstate tvorí referenčný model z pohľadu výkonu. Je napísaný v jazyku C a využíva virtuálny stroj BPF na filtrovanie paketov. Ukážka použitia nástroja tcpdump:

tcpdump –r "cesta k súboru" tcp port 80 – vyfiltruje všetky pakety, ktoré nemajú zdrojový alebo cieľový port protokolu TCP rovný hodnote 80

5 Knižnica NetBee

NetBee (5) je knižnica zameraná na spracovanie paketov, ktorá podporuje analyzovanie a filtrovanie paketov, dekódovanie paketov. Je vyvíjaná týmom NetGroup[9] na Politecnico di Torino (Taliansko) ako open source pod licenciou BSD. Poskytuje množinu modulov, ktoré môžu aplikácie využiť na spracovanie sieťovej prevádzky namiesto vytvárania vlastného kódu. Knižnica sa skladá z niekoľkých komponent ako napríklad NetPDL (jazyk pre popis protokolov) a NetVM (virtuálny stroj na spracovanie paketov). Je vyvíjaná tým istým tímom, ktorý vytvoril WinPcap. Architektúra WinPcap je podľa tvorcov zastaraná a nereflektuje dnešné potreby, predovšetkým jednoduché pridávanie rozšírení a nových modulov pre iné typy spracovania paketov. Z tohto dôvodu vznikol projekt NetBee postavený na úplne novej architektúre (viď podkapitolu 5.1).



Obrázok 5: Architektúra knižnice NetBee

Knižnica NetBee je v súčasnosti v experimentálnej fázy vývoja. Funkčné verzie slúžia na potvrdenie konceptov a výmenu ideí s výskumnou komunitou. Nás zaujíma predovšetkým architektúra knižnice a výkonnostné výsledky virtuálneho stroja určeného priamo na spracovanie paketov. Od autorov projektu sme získali zdrojové kódy s cieľom preložiť si vlastné knižnice a spustiteľné súbory pre platformu Linux a vyskúšať si niektoré ukážkové príklady. Preklad samotnej knižnice podľa presného návodu prebehol bez väčších problémov, avšak nepodarilo sa

[9] http://netgroup.polito.it/

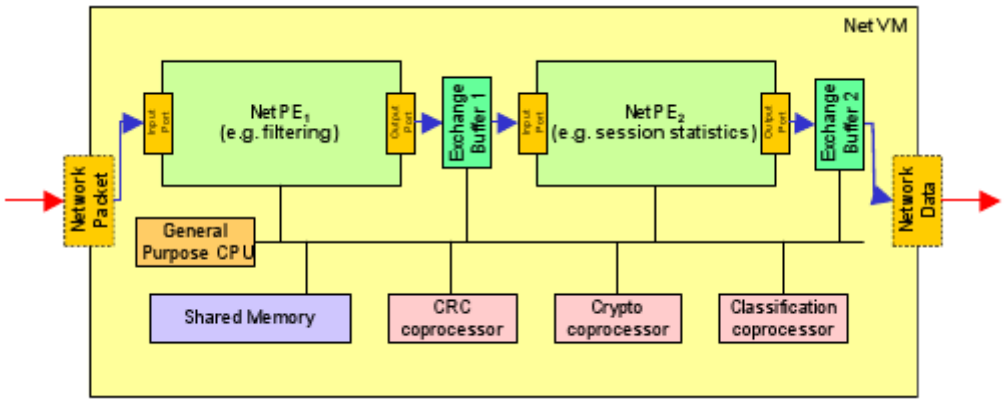
preložiť všetky ukážkové príklady a nástroje. Tu sa naplno prejavila experimentálna povaha projektu v podobe nekonzistencie API. Súčasťou projektu je aj nástroj nbeedump (viď podkapitolu 5.2), podobný programu tcpdump (viď kapitolu 4), ktorý sa nám podarilo preložiť a použijeme ho na porovnanie výkonu v závere diplomovej práce. Zdrojové súbory sú na priloženom CD v zložke *Netbee/src*, binárne súbory sú v zložke *Netbee/bin/Linux*.

5.1 Architektúra a vlastnosti

Knižnica NetBee je implementovaná v jazyku C++. Je dostupná pre platformu x86 Windows, Linux a Mac OS X, ale hlavná vývojová platforma je Windows. Preložené binárne súbory fungujú aj na x86-64 Linux po doinštalovaní potrebných 32 bitových knižníc. JIT pre VM je dostupný iba pre 32 bitové verzie operačných systémov, s tým, že podľa slov samotných autorov obsahuje ešte veľa chýb, predovšetkým pre platformu Mac OS X. Všeobecná štruktúra knižnice je zachytená na obrázku 5.

5.1.1 Triedy exportované z NetBee

NetBee definuje množinu tried použiteľnú na spracovanie sieťovej prevádzky. Všetky moduly pod NetBee API (viď obrázok 5) sú triedy exportované k užívateľovi. Ten ich môže použiť vo svojej aplikácii na požadovaný typ spracovania paketov. Tieto triedy vyžadujú k svojej činnosti ďalšie moduly zapuzdrené v NetBee, ktoré nie sú exportované k užívateľovi (viď obrázok 5 pravý dolný roh).



Obrázok 6: Architektúra NetVM

5.1.2 NetBee virtuálny stroj

NetBee virtuálny stroj (6) je virtuálny sieťový procesor optimalizovaný na spracovanie sieťovej prevádzky ako napríklad filtrovanie paketov. Podobne ako Java VM virtualizuje CPU, NetVM virtualizuje sieťový procesor. NetVM má poskytnúť unifikovanú vrstvu pre sieťové úlohy (filtrovanie, počítanie paketov, ...) vykonávané rôznymi sieťovými aplikáciami (firewally, monitory siete, ...) tak, aby mohli byť spustené na každom sieťovom zariadený počínajúc sofistikovanými smerovačmi až po jednoduché prepínače. Program pre NetVM je platformovo nezávislý,

pretože môže byť za behu pomocou interpreta (prípadne pomocou JIT kompilácie) preložený do natívneho kódu danej hardvérovej platformy. Vďaka tejto flexibilite môže byť NetVM implementované aby fungovalo na rôznych zariadeniach, umožňujúc programovať tieto zariadenia tretími stranami.

NetVM nasleduje princípy (viď obrázok 6), ktoré inšpirovali architektúru sieťových procesorov, odlišnú od všeobecne zameraných virtuálnych strojov. To sa odráža aj na množine podporovaných inštrukcií, ktorá je redukovaná v porovnaní so všeobecnými VM, na druhej strane však zameraná na manipuláciu a spracovanie paketov (klasifikácia, podpora asociatívnych polí, výpočet CRC, kryptovanie, ...). NetVM sa skladá z tzv. NetPE. Jedná sa o virtuálne procesory (zásobníkové), na ktorých bežia programy v assemblery NetVM. Vykonávajú jednoduché funkcie ako napr. filtrovanie paketov. NetPE v rámci NetVM môže byť niekoľko, každý zameraný na inú funkciu, pričom môžu byť reťazené alebo paralelizované do zložitejších štruktúr. Keďže NetVM môže byť potenciálne mapované na sieťové procesory, použitie vysokoúrovňovej správy pamäte v podobe GC nie je vhodné. Naviac pamäť je staticky alokovaná počas inicializačnej fázy.

V súčasnosti beží NetVM na troch heterogénnych hardvérových platformách: Intel ia32 (tradičné x86), Cavium Octeon a Xelerated X11 sieťový procesor. Na x86 je momentálne všetko vykonávané sekvenčne v jednom vlákne. Ako náhle vstúpi do NetVM paket, kód priradený jednotlivým NetPE je volaný sekvenčne, kým nie je dosiahnuté výstupné rozhranie.

```
<protocol name="ethernet" longname="Ethernet 802.3" comment="Ethernet
 DIX has recently been included in 802.3" showsumtemplate="ethernet">
    <format>
        <fields>
            <field type="fixed" name="dst" longname="MAC Destination"
             size="6" showtemplate="MACaddressEth"/>
            <field type="fixed" name="src" longname="MAC Source"
             size="6" showtemplate="MACaddressEth"/>
            <field type="fixed" name="type" longname="Ethertype -
             Length" size="2" showtemplate="eth.typelength"/>
        </fields>
    </format>
    <encapsulation>
        <switch expr="buf2int(type)">
            <case value="0x800">
                <nextproto proto="#ip"/>
            </case>
        </switch>
    </encapsulation>
</protocol>
```

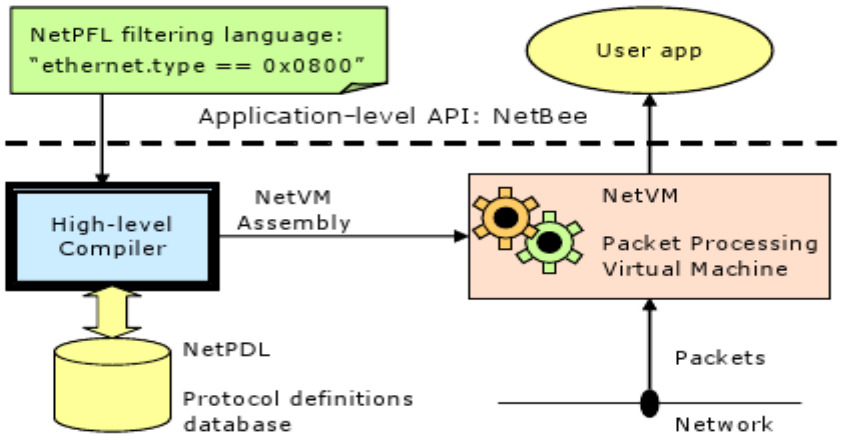
Výpis 6: Ukážka databáze protokolov v jazyku NetPDL

5.1.3 Virtuálne sieťové zdroje

Objekty sieťových zdrojov sú abstraktné zariadenia pre virtualizáciu skutočných sieťových zdrojov ako napríklad sieťových rozhraní, súborov, iných aplikácií. Jeden z problémov iných knižníc zameraných na spracovanie paketov je volanie rôznych funkcií na rôzne zdroje. Tento modul rieši tento problém a umožňuje programátorom pracovať s generickými sieťovými zdrojmi, nechávajúc komplexnosť riadenia skutočného zdroja na príslušnom NetBee objekte. Verejné rozhrania virtuálnych sieťových zdrojov sú vytvorené tak, aby mohli byť prepojené medzi sebou priamo alebo cez NetVM objekt (napr. prvý zdroj ako továreň paketov a druhý zdroj ako cieľ paketov v podobe súboru na disku).

5.1.4 Databáza protokolov

Niektoré aplikácie potrebujú poznať formát paketov aby mohli vykonávať svoju úlohu ako napríklad filtre paketov (potrebujú presne lokalizovať polia, ktorých hodnoty musia byť overené). Aby sme sa vyhli vytváraniu vlastného popisu protokolov pre každú aplikáciu zvlášť, vznikol jazyk NetPDL. Jazyk je založený na XML a umožňuje definovať štruktúru hlavičiek protokolov. Databáza protokolov je databáza definícii protokolov podľa jazyka NetPDL. Akonáhle sa zmení popis nejakého protokolu v databáze, alebo je pridaný nový protokol, všetky moduly sú okamžite schopné spracovať nový typ protokolu. Výpis 6 zobrazuje ukážku databáze protokolov, konkrétne Ethernet.



Obrázok 7: Princíp fungovania NetBee

5.2 Nástroj nbeedump

Tento nástroj v podstate predstavuje redukovanú verziu tcpdumpu (viď kapitolu 4) implementovanú pomocou NetBee knižnice. Na tomto nástroji si popíšeme princíp fungovania NetBee. Najskôr si ukážeme príklad použitia a potom si popíšeme čo sa vlastne vo vnútri NetBee deje.

Ukážka použitia:

./nbeedump -netpdlfile netpdl-min.xml -q -r /home/testVM/file800MB.pcap "tcp.sPort==80 or tcp.dPort==80"

Z ukážky je asi dopredu jasné, že budeme čítať súbor *file800MB.pcap,* z ktorého príjmeme len pakety so zrojovým alebo cieľovým portom 80 protokolu TCP. Princíp fungovania je dobre zachytený na obrázku 7. Kompilátor na vstupe obdrží reťazec resp. filter v jazyku NetPFL v našom prípade „*tcp.sPort==80 or tcp.dPort==80*“. Štruktúru hlavičiek protokolov si načíta z databáze protokolov. V súčasnosti vie, ktoré polia ma porovnávať a kde ich ma hľadať. Na základe týchto informácií vytvorí program v assemblery NetVM, ktorý predá samotnej NetVM. Tá spracováva pakety podľa požiadavkou a výsledky posiela do užívateľskej aplikácie v našom prípade nbeedump. Výpis 7 zobrazuje ukážku zdrojového kódu resp. hlavné cyklu nbeedump, ktorý číta pakety zo súboru a posiela ich do NetVM na spracovanie.

```
while (1) {
   RetVal = pcap_next_ex(fp, &PktHeader, &PktData); //get packet

   if (RetVal == -2)
      break;            // capture file ended
   if (RetVal < 0) {
      printf("Cannot read packet: %s\n", pcap_geterr(fp));
      return nbFAILURE;
   }
   // Timeout expired
   if (RetVal == 0)
      continue;

   if (ConfigParams.FilterString != NULL){
      //process packet with NetVM
      if(PacketEngine->ProcessPacket(PktData, PktHeader->len) ==
         nbSUCCESS)
         WritePacketFunct(PktData,PktHeader->len, &decoderInfo, &ts);
   }
   else
      WritePacketFunct(PktData, PktHeader->len, &decoderInfo, &ts);

   count++;
   // Check if the user wanted to capture max N packets
   if ((ConfigParams.NPackets != 0) && (PacketCounter ==
       ConfigParams.NPackets))
      break;
}
```

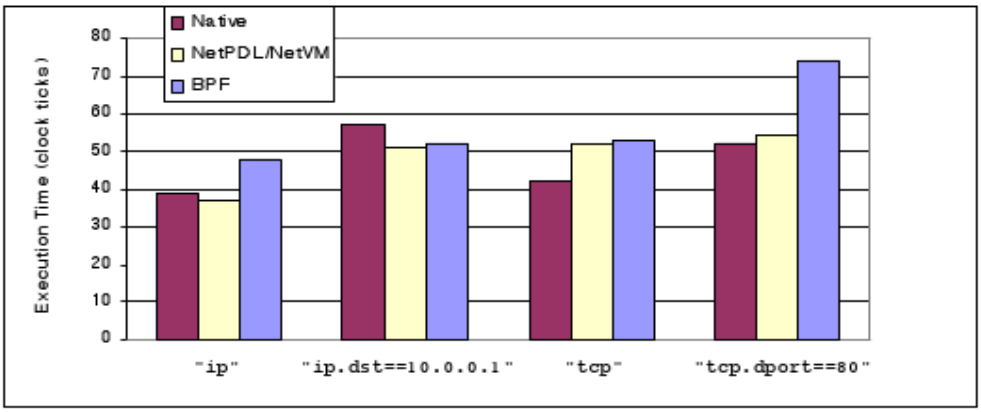
Výpis 7: Hlavný cyklus filtra v NetBee

Počas prvých testov sa prejavila chybovosť JIT pre platformu Linux, kedy pri použití úplnej databáze s popisom protokolov dochádzalo k chybám prístupu do pamäte (segmentation false).

Aby sme mohli program používať, museli sme pristúpiť k používaniu minimálnej verzie databáze.

5.3 Zhodnotenie výkonu (2008)

Samotní autori knižnice vykonali niekoľko testov (7) aby konfrontovali NetBee so súčasnými najviac rozšírenými riešeniami. Porovnávali vykonávanie štyroch rôznych filtrov za použitia úplnej databáze protokolov, s riešeniami založenými na BPF a jazyku C (natívny filter). Testy boli uskutočnené na 3GHz Intel pentium 4. NetVM a BPF programy boli vykonávané pomocou JIT kompilácie. Natívny filter bol naprogramovaný v jazyku C a preložený pomocou MS Visual C++ 2005 kompilátorom. Každý test zahrňoval spracovanie 10000 paketov a bol spustený 12 krát. Výsledky boli spriemerované, pričom najlepší a najhorší výsledok pre dané meranie bol vylúčený. Celkové vyhodnotenie je prehľadne spracované na obrázku 8.



Obrázok 8: Porovnanie výkonu NetBee (2008)

6 Virtuálny stroj Parrot

Parrot (8) je virtuálny stroj navrhnutý na efektívne kompilovanie do bytekódu a interpretovanie dynamických jazykov. Je vyvíjaný neziskovou organizáciou Parrot Foundation ako open source. Medzi hlavných podporovateľov projektu patria: ActiveState, BBC, NLNet Foundation a Mozilla Foundation. V súčasnosti hostí Parrot veľké množstvo predovšetkým dynamických jazykov v rôznom stupni vývoja: Javascript, Ruby, PHP, Python, Perl 6, ...

Parrot pôvodne vznikol ako behové prostredie pre Perl 6. Narozdiel od Perlu 5, je v novej verzii Perlu jasne oddelený kompilátor do bytekódu od behového prostredia (VM). Samotný názov reflektuje zámer vytvoriť virtuálny stroj nie len pre Perl 6, ale aj pre mnoho ďalších (dynamických) jazykov.

Parrot je bežne dostupný v operačných systémoch typu Linux. Na jeho stiahnutie použite inštalačný manažér dostupný vo vašej distribúcii (yum Fedora, aptitude Ubuntu, ...). Pre účely diplomovej práce sme použili postup[10] stiahnutia najnovších zdrojových súborov zo stránky Parrotu a manuálneho preloženie a inštalácie. Okrem základnej inštalácie Parrotu budeme potrebovať aj vývojárske nástroje, ktoré použijeme pri tvorbe samotného filtra. Podrobný návod na stiahnutia a inštaláciu Parrotu je uvedený na domovských stránkach projektu. Napriek tomu si jednoducho popíšeme počiatočnú fázu prekladu, lebo počas implementácie filtra boli zistené nedostatky v Parrote, ktoré sme museli odstrániť ručne. V časti 6.1.5 sa môžeme dočítať o spôsobe volaní externých C funkcií a predávaní signatúr. Parrot má napevno definované všetky prípustné signatúry, pričom nepozná jednu signatúru (*lt*) potrebnú pre naša účely. Túto signatúru pridáme nasledujúcim spôsobom, najskôr musíme nakonfigurovať Parrot pred samotným prekladom: *perl Configure.pl*. Konfigurácia vygeneruje v zložke *./src/* súbor *call_list.txt*. Do tohto súboru pridáme našu signatúru a pokračujeme v preklade Parrotu podľa štandardného postupu.

6.1 Architektúra a vlastnosti

Virtuálny stroj Parrot je implementovaný v jazyku C. Oficiálne je dostupný pre platformu x86 Windows, Linux a Mac OS X, pričom hlavná vývojová platforma je Linux. Bez problému sa nám ho však podarilo preložiť aj na platforme x86-64 Linux. Parrot je virtuálny stroj založený na registroch (register based), narozdiel od súčasných VM ako Java VM, CLR, Perl 5 VM, ktoré sú zásobníkové. Vývojári boli motivovaný predovšetkým rozdielmi medzi dynamickými a statickými jazykmi a inšpirovaný ďalšími virtuálnymi strojmi Lua[11] a Dis[12] (obidva sú register based).

[10] Firma Linuxbox.cz používa špecifickú distribúciu Linuxu založenú na CentOS, kde nie je možné stiahnuť Parrot a neskôr aj LLVM pomocou inštalačného manažéra.
[11] http://en.wikipedia.org/wiki/Lua_(programming_language)#Internals
[12] http://en.wikipedia.org/wiki/Dis_virtual_machine

Orientácia Parrotu na dynamické jazyky sa prejavila vo forme podpory automatickej správy pamäte v podobe GC, podpory vlákien a výnimiek. Parrot obsahuje aj dátový typ pointer, avšak nepodporuje pointerovú aritmetiku.

V čase začiatku práce s Parrotom vo verzii 1.6.0 obsahoval projekt aj JIT kompiláciu. Pri dokončení práce s Parrotom, kedy bola dostupná už verzia 2.0.0, bolo zistené odobranie JITu od verzie 1.7.0. Táto informácia mala zásadný dopad na diplomovú prácu. V poslednej fáze vývoja filtra, ktorý si prešiel niekoľkými štádiami, sme ho prispôsobili verzii Parrotu 2.0.0. Tým sme sa nechtiac pripravili o možnosť otestovať riešenie za pomoci JITu. Aj keď by bolo možné filter spätne prispôsobiť starším verziám Parrotu obsahujúcich JIT, táto varianta bola zamietnutá. Stávajúci JIT bol úplne odobraný a na vývojárskych stránkach Parrotu (9) je popísaný zámer použiť resp. vyskúšať iné riešenia (libJIT, LLVM JIT, GNU Lighting, nanoJIT). Na základe týchto dôvodov bude riešenie založené na VM Parrot otestované v závere diplomovej práve bez podpory JIT kompilácie.

6.1.1 Formát inštrukcií

Parrot v súčasnosti akceptuje inštrukcie resp. príkazy v štyroch rôznych formátoch:

1. PIR (Parrot Intermediate Representation) – je natívny nízko úrovňový jazyk, v podstate sa jedná o assembler s niektorými vysoko úrovňovými vlastnosťami ako použitie operátorov, zjednodušená syntax (syntax sugar) pre funkcie a metódy, automatická alokácia registrov, atď. Je navrhnutý tak, aby bol čitateľný pre ľudí a zároveň ľahko generovateľný kompilátorom. Samotné knižnice Parrotu a niektoré kompilačné nástroje sú napísané priamo v PIR. Tento jazyk bol použitý aj pri tvorbe filtra.
2. PASM (Parrot Assembly) – je jazyk o úroveň nižšie ako PIR, stále čitateľný pre ľudí a generovateľný kompilátormi. Neposkytuje však menné registre, direktívy *.sub* a *.end* (nahradené návestiami),...
3. PAST (Parrot Abstract Syntax Tree) – umožňuje Parrotu akceptovať na vstupe abstraktný syntaktický strom, čo je výhodné pri písaní vlastného kompilátora.
4. PBC (Parrot Bytecode) – všetky spomínané formy inštrukcií sú automaticky vo vnútri Parrotu konvertované práve do PBC. Jedná sa o strojový jazyk VM Parrot, ktorý nie je čitateľný pre ľudí. Na rozdiel od ostatných foriem môže byť vykonávaný okamžite. Je platformovo nezávislý.

6.1.2 Registre a základné dátové typy

Parrot je, ako už bolo spomínané v úvode tejto kapitoly, založený na registroch. To znamená, že podobne ako reálne hardvérové CPU má k dispozícii rýchle jednotky na ukladanie, nazývané registre. Parrot ponúka štyri základné dátové typy resp. registre:

1. Integers (I) – celé čísla – majú veľkosť slova (word) na stroji na ktorom beží Parrot
2. Numbers (N) – desatinné čísla – sú mapované na natívne desatinné typy

3. String (S) - reťazce
4. PMC (P) – polymorfické kontajnery – reprezentujú komplexné dátové štruktúry a typy (polia, asociatívne polia, ...)

Existujú dva spôsoby ako sa odkazovať na registre:

1. Pomocou premenných registrov resp. názvov zložených zo znaku $, prvého písmena typu registra a čísla počínajúc 0 ($I0, $I5, $P1, ...).
2. Použitím pomenovaných premenných deklarovaných pomocou *.local* (.local pmc foo).

Názov ($I1) registra však nemusí odpovedať číslu registra, ktorý je použitý interne. Kompilátor Parrotu mapuje názvy na registre s ohľadom na výkon a pamäť. Jedinú záruku, ktorú Parrot poskytuje, je použitie toho istého úložného miesta pre rovnaký názov ($I1) v danej funkcii resp. metóde.

6.1.3 Polymorfické kontajnery

V striktnom zmysle slova sú PMC napísané v jazyku C a prekladané kompilátorom do strojového kódu. Avšak Parrot používa špeciálny kompilátor na konvertovanie PMC napísaných v „skriptovacom“ jazyku podobnom C do štandardného C. Všetky konštrukcie jazyka C sú platné aj v skriptovacom jazyku, ktorý naviac prináša niekoľko doplnkov na uľahčenie bežných úloh. Parrot je interne napísaný v jazyku C podľa normy ISO C89 kvôli maximálnej prenositeľnosti. Avšak PIR a jazyky postavené na VM Parrot sú typicky objektovo orientované (alebo majú nejaké OO vlastnosti). Samotné PMC sú ako triedy, majú dáta (atribúty), funkcie (metódy) a môžu dediť z iných PMC. C je síce nízko úrovňový a prenositeľný, čo je žiadané, ale nepodporuje OO vlastnosti. Práve kvôli tomu vznikol PMC skriptovací jazyk a kompilátor do C89.

Pomocou PMC Parrot implementuje všetky dátové typy zložitejšie ako integer, number a string. Z nášho pohľadu sa jedná o abstraktný dátový typ, ktorý môže reprezentovať reťazec, číslo, kód, pole, paket, atď. Táto miera abstrakcie nám umožňuje jednať s PMC ako so štandardným API pre manipuláciu s dátami. Medzi ďalšie vlastnosti PMC patrí:

- obsahuje oboje, stav (dáta resp. atribúty) aj chovanie (vtable funkcie resp metódy)
- môže byť postavené z nízkoúrovňových rolí – napr. priamy prístup do pamäte
- môžu byť zakomponované do binárnych súborov Parrotu alebo načítané dynamicky v prípade potreby (použité vo filtri)

6.1.4 Vtables

Spomínanú mieru abstrakcie PMC (viď 6.1.3) nám umožňuje práve vtables. Jedná sa o množinu pointerov na funkcie, ktoré určujú, ako sa bude PMC chovať pri rôznych operáciach. PMC si teda môžeme predstaviť ako abstraktnú triedu, ktorá definuje množinu metód na implementáciu. To nám umožňuje oddeliť bežné operácie ako sčítanie, rozdiel, násobenie, ... spoločné všetkým bežným jazykom od špecifického chovania vyžadovaného každým jazykom zvlášť.

Vtable je štruktúra, ktorá obsahuje pointeri na funkcie. Každé PMC obsahuje pointer na takúto štruktúru, ktorá implementuje jeho chovanie. Čiže ak chceme zistiť veľkosť PMC, zavoláme funkcie *length* na PMC, Parrot vyhľadá vo vtable (na ktorú ukazuje) slot pre veľkosť, načíta si príslušný pointer a zavolá funkciu na zistenie veľkosti so sebou ako parametrom.

(pmc->vtable->length)(pmc);

```
/* foo.c */
/* specify the function prototype */
#ifdef __WIN32
     __declspec(dllexport) void foo(void);
#else
     void foo(void);
#endif

void foo(void) {
     printf("Hello Parrot!\n");
}

#foo.pir
.sub main :main
.local pmc lib, func

      # načíta zdieľanú knižnicu
      lib = loadlib "hello" # no extension, .so or .dll is assumed
      # získa odkaz na požadovanú funkciu z načítanej knižnice a
      # predá parrotu signatúru funkcie „void“ (žiadne argumenty)
      func = dlfunc lib, "foo", "v"
      # invoke
      func()
.end
```

Výpis 8: Predanie signatúry Parrotu

Pri implementácii vlastného PMC teda aj vtable je potrebné presne dodržiavať prototypy jednotlivých vtable funkcií. Ak nám niektorá funkcia nevyhovuje resp. ju nebudeme potrebovať, môžeme ju jednoducho vynechať[13]. Keďže vtable funkcií je obmedzený počet, nie vždy pokryjú námi požadované chovanie. Našťastie si môžeme chovanie doplniť v podobe metód. Pri volaní metód PMC musíme názov volanej metódy dať do úvodzoviek.

*METHOD inspect(STRING *what :optional, int got_what :opt_flag) {...}*

6.1.5 Rozhranie pre natívne volania

NCI umožňuje Parrotu volať funkcie z C knižníc bez toho, aby bolo potrebné napísať nejaký kód v C. Nie je navrhnuté ako univerzálne rozhranie, ktoré zvládne každú knižnicu so všelijakými bizarnými parametrami (v týchto prípadoch sa napísaniu C kódu nevyhneme) ale pokryje

[13] Nesmie byť v kóde vôbec, ani v podobe prázdnej funkcie.

väčšinu jednoduchých prípadov. Pri použití NCI Parrot automaticky zaobalí C funkciu a prezentuje ju ako prototyp funkcie, ktorá dodržiava všetky konvencie volaní. Takto obalenú funkciu môžeme normálne volať ako každú inú.

NCI používa platformovo natívny dynamický mechanizmus načítania funkcií (napríklad dlopen/dlsym na Unix a LoadLibrary/GetProcAddress na Win32) na získanie pointeru na funkciu. Potom dynamicky vygeneruje „obal“ na základe signatúry (prototypu) funkcie. Keďže neexistuje obecný platformovo nezávislý spôsob ako zistiť signatúru funkcie (napríklad nie vždy sú k dispozícii hlavičkové súbory), musí byť Parrotu signatúra predaná ručne (viď výpis 8).

6.1.6 Základy jazyka PIR

V tejto podkapitole si popíšeme základy jazyka PIR. Zameriame sa predovšetkým na použité inštrukcie vo filtri, s ktorými sme sa počas tvorby programu stretli. Nebudú tu popísané premenné a práca s registrami, tie boli spomenuté v podkapitole 6.1.2. PIR ako už bolo spomínané podporuje veľa konštrukcií a operátorov, na ktoré sme zvyknutý z vyšších programovacích jazykov (operátor priradenia =, aritmetické operátory +, -, *, /, operátory bitového posunu <<, >>, logické bitové operátory &, |).

Komentáre

Komentáre začínajú symbolom # a pokračujú až do konca riadku ako sme zvyknutý z iných jazykov. Dokumentačný text začínajúc znakom = a končiac značkou *=cut* je v PIR braný tiež ako komentár.

```
# Toto je jednoriadkový komentár, interpreter PIRu ho ignoruje.

=head2
Toto je viacriadkový komentár, interpreter PIRu ho ignoruje. Zároveň slúži pre vytváranie
a generovanie dokumentácie.
=cut
```

Návestia

Návestia pripájajú k riadkom kódu mená, takže ostatné príkazy sa môžu na ne odkazovať. Návestia súvisia priamo s **riadiacimi štruktúrami.** PIR nemá priamo štruktúry pre prepínač, cykly, namiesto toho obsahuje malú množinu podmienených a nepodmienených skokov, ktorými si môžeme všetky požadované štruktúry naimplementovať.

```
goto nepodmieneny_skok # goto je inštrukcia nepodmieneného skoku
    say "nikdy sa nevypíše" # príkaz say vypisuje predaný reťazec na stdout
nepodmieneny_skok: # nepodmieneny_skok je názov návestia
    say "toto sa vypíše"

if podmienka goto mozno_skok # if podmienka goto je inštrukcia podmieneného skoku
    say "môže sa vypísať" # vypíše sa ak nie je podmienka pravdivá
```

```
mozno_skok:
    say " môže sa vypísať po skoku" # vypíše sa ak je podmienka pravdivá
```

Podprogramy

Podprogramy v PIR začínajú direktívou *.sub* a končia direktívou *.end*. Na predanie parametru sa používa direktíva *.param*, ktorá sa použitím podobá pomenovaným premenným (viď podkapitolu 6.1.2). Pri spustení programu Parrot vykoná prvú funkciu, ktorú nájde. Toto základné chovanie môžeme upraviť modifikátorom *:main* za názvom funkcie, ktorý Parrotu povie aby takto označenú funkciu vykonal ako prvú.

```
.sub vypisText          # definícia funkcie
    .param string text  # parameter funkcie
    say text
.end
vypisText(“Ahoj svet!”) # použitie funkcie
```

Direktívy

Direktívy sa podobajú na normálne inštrukcie ale začínajú znakom bodka (.). Niektoré špecifikujú akcie, ktoré sa realizujú počas kompilácie, iné reprezentujú komplexné operácie, ktoré si vyžadujú vygenerovanie viacerých inštrukcií.

```
.const int DEBUG = 0   # direktíva na špecifikovanie konštanty
.loadlib 'packet'      # direktíva na načítanie dynamického PMC
.param .local .sub .end  # s týmito direktívami sme sa už stretli
```

Agregované dátové typy

Parrot podporuje rôzne typy polí, medzi ktorými samozrejme nechýbajú polia indexované pomocou celých čísel a asociatívne polia indexované reťazcami. Tieto zložité dátové typy sú realizované pomocou PMC.

```
$P0    = new "ResizablePMCArray" # vytvorí pole s variabilnou veľkosťou
$P0[0] = 10                      # nastaví prvý element na 10
$I0    = $P0[0]                  # vráti prvý element
$P0 = 2                          # nastaví veľkosť poľa na 2
push $P0, 'banana'               # vloží element nakoniec poľa
```

Ďalšie inštrukcie

V tejto časti si popíšeme ostatné inštrukcie použité v programe. Jedná sa predovšetkým o inštrukcie na inicializovanie pomenovaných PMC, na načítanie resp. importovanie modulov, na pridanie atribútov existujúcim PMC.

```
packet = new "Packet"          # inicializuje pomenovanú premennú na typ Packet
```

load_bytecode "./Decoder.pbc" # importuje modul Decoder.pbc
setprop packet, "tcp", $P0 # pridá PMC atribút „tcp“ a nastaví jeho hodnotu na $P0
$P1 = getprop "tcp", packet # vráti hodnotu atribútu “tcp” premennej packet ak existuje

6.2 Implementácia filtra

Implementácia filtra je založená na princípoch popísaných v kapitole 1 (použitie mmapu, ...). Prešla si niekoľkými štádiami a kvôli dôvodom, ktoré si popíšeme za chvíľku, si vyžiadala prehodnotenie použitých techník a prepísanie skoro hotového filtra. Všetky zdrojové súbory sú na priloženom CD v zložke *Parrot/src*.

Na začiatku implementácie bol zvolený postup inicializácie a spúšťania Parrotu z C. Jedná sa o princíp nemálo podobný riešeniam založeným na BPF alebo NetBee, kedy je VM používané (zapuzdrené) v jazyku C/C++. Nesie to so sebou ďalšiu výhodu v podobe možnosti využívať všetky knižnice daného jazyka ako napríklad mmap. Parrot má pre túto technológiu vlastný názov „Parrot embedding system“ (ďalej len PES).

```
void* mapFile2Memory(char* pcap_file) {
    ...
    start_time = time(0);
    void* address = mmap(0, file_size, PROT_READ, MAP_PRIVATE |
                         MAP_POPULATE |  MAP_NORESERVE, fpcap, 0);
    printf("mmap time: %d s\n", time(0) - start_time);

    if (address == MAP_FAILED) {
       close(fpcap);
       fprintf(stderr, "Can`t map file: %s to memory!\n", pcap_file);
       perror("mmap");
       return 0;
    }
    close(fpcap);
    return address;
}
```

Výpis 9: Namapovanie súboru do pamäte

Najskôr boli vykonané jednoduché testy, ktoré mali posúdiť vhodnosť tejto technológie pre náš účel. Počiatočné testy prebehli úspešne a toto riešenie sa javilo ako správna cesta. Jedinou prekážkou bolo spočiatku nekonzistentné API pre PES (prototypy funkcií v dokumentácii neodpovedali skutočnosti). Počas práce sme postupne narazili na niekoľko chýb, ktoré sme buď opravili alebo nejakým spôsobom obišli. Ďalšie problémy na seba nenechali dlho čakať. Jednalo sa predovšetkým o reprezentáciu namapovaného súboru a paketu v Parrote a predanie pointeru z C do Parrotu. Tieto problémy boli vyriešené implementáciou vlastných PMC (viď podkapitoly 6.2.1 a 6.2.2). Tieto PMC však neboli načítané dynamicky, ale kvôli nemožnosti predať vlastné PMC z C do Parrotu, boli začlenené priamo do jadra Parrotu. Tento prístup bol veľmi neefektívny, pri každej zmene PMC bolo potrebné preložiť znova celý Parrot. Nakoniec sa podarilo

implementovať funkčný filter, ktorý úspešne prešiel počiatočnými testami na malom súbore. Avšak neskôr pri teste na veľkom (800 MB) súbore bola odhalená závažná chyba v podobe úniku pamäte. Táto chyba bola samozrejme oznámená komunite Parrotu a nezostávalo nič iné ako čakať na opravu. Avšak ani po dvoch mesiacoch čakania nebola chyba odstránená. Stávajúce riešenie bolo zamietnuté a hľadali sa nové postupy ako implementovať filter. Riešenie (so zachovaním princípov z kapitoly 1) nakoniec ponúklo NCI (viď podkapitolu 6.1.5), ktoré umožňuje volať funkcie z C knižníc priamo v PIR.

Aby sme nemuseli v Parrote používať veľké množstvo C funkcií, implementovali sme za týmto účelom jednoduchú zdieľanú knižnicu *libMapFile2Memory.so.* Jej účelom je namapovať súbor do pamäte (viď výpis 9) a zistiť jeho veľkosť.

```
/* Nacita zo suboru paket. */
METHOD get_packet(PMC *packet_p) {
       Parrot_PCAPFileReader_attributes * const data =
             PARROT_PCAPFILEREADER(SELF);
       unsigned int packet_len;
       void *packet_pointer;

       if (data->offset == -1 || data->file_size == -1) {
             RETURN(INTVAL -1);
       }
       if (data->offset >= data->file_size) {
             RETURN(INTVAL -2);
       }

       //prvy precitany oktet je vyznamovo najnizsi
       packet_len = (((unsigned char *)data->pointer)[data->offset+8]);
       packet_len |= (((unsigned char *)data->pointer)[data->offset+9])<<8;
       packet_len |= (((unsigned char *)data->pointer)[data->offset+10])<<16;
       packet_len |= (((unsigned char *)data->pointer)[data->offset+11])<<24;

       data->offset += 16; //preskoci hlavicku
       packet_pointer = ((unsigned char *)data->pointer) + data->offset;
       data->offset += packet_len; //preskoci packet

       VTABLE_set_pointer(INTERP, packet_p, packet_pointer);
       VTABLE_set_integer_native(INTERP, packet_p, (INTVAL)packet_len);

       RETURN(INTVAL 0);
}
```

Výpis 10: Načítanie paketu zo súboru v PMC packet

6.2.1 Packet PMC

V tejto časti si okrem samotného PMC popíšme aj postup ako pridať dynamické[14] PMC do Parrotu. Packet PMC reprezentuje načítaný paket (pointer na pole prvkov typu char) zo súboru a umožňuje nad ním vykonať potrebné[15] funkcie:

[14] Dynamicky načítané Parrotom podľa potreby.
[15] Jedná sa o operácie, predovšetkým pointerovú aritmetiku, ktoré PIR neumožňuje.

1. Nastavenie a prečítanie offsetu:
   - `METHOD set_offset(INTVAL value)`
   - `METHOD get_offset()`
2. Prečítanie práve 8/16/32 bitov z poľa a vrátenie prečítanej hodnoty ako INTVAL (celočíselný dátový typ Parrotu):
   - `METHOD get_8_bit(INTVAL ofset)`
   - `METHOD get_16_bit(INTVAL ofset)`
   - `METHOD get_32_bit(INTVAL ofset)`

```
.sub decodeTCP
      .param pmc packet_p
      .param int offset_i
      .local int sPort_i, dPort_i

      #0-15 source port
      sPort_i = packet_p.'get_16_bit'(offset_i)
      #16-31 dest port = 2 bytes
      $I0 = offset_i + 2
      dPort_i = packet_p.'get_16_bit'($I0)

      $P0 = box 1
      #ked najde tcp nastavi, ze ho nasiel
      setprop packet_p, "tcp", $P0

      $P0 = box sPort_i
      #doplni ako vlastnosti tcp porty
      setprop packet_p, "tcp_sPort", $P0
      $P0 = box dPort_i
      setprop packet_p, "tcp_dPort", $P0

.end
```

Výpis 11: Dekódovanie protokolu TCP v Parrote

Ostatné operácie sú realizované pomocou vtable funkcií (predanie pointeru, nastavenie veľkosti paketu, prečítanie veľkosti paketu, ...). V definícii PMC musíme použiť kľúčové slovo dynpmc, ktoré hovorí, že námi vytvorené PMC bude načítané dynamicky. Takto pripravené PMC *packet.pmc* prekonvertujeme pomocou nástrojov Parrotu do C a vytvoríme z neho zdieľanú knižnicu. Keďže sa jedná o sériu krokov, vytvoríme si za týmto účelom *makefile*. Parrot obsahuje nástroj, ktorý prevedie jednoduchý *makefile.in* zhotovený zo šablóny na reálny *makefile*, ktorý preloží a nainštaluje dynamické PMC:

perl gen_makefile.pl[16] *Makefile.in Makefile*

[16] */usr/local/lib/parrot/1.6.0/tools/dev/* - cesta sa môže líšiť v závislosti na použitej platforme (distribúcii Linuxu) a verzii Parrotu

make –f Makefile
make –f Makefile install

6.2.2 PcapFileReader PMC

PcapFileReader PMC reprezentuje namapovaný súbor (pointer na pole prvkov typu char) vo formáte knižnice PCAP (viď kapitolu 1) do operačnej pamäte. Hlavnou funkciou tohto PMC je čítať jednotlivé pakety zo súboru a ukladať ich do predaného parametra PMC packet. Výpis 10 zobrazuje ukážku načítania paketu zo súboru.

```
$P0 = box 0
#podstata dekodovania je v glob premennych resp. atributoch
#pred dekodovanim hodnoty inicializujem na 0
setprop packet_p, "tcp", $P0
setprop packet_p, "tcp_sPort", $P0
setprop packet_p, "tcp_dPort", $P0

loop_read_packet:
      $I0 = pcap_file_reader_p.'get_packet'(packet_p)
      if $I0 < 0 goto loop_read_packet_end

      decode(packet_p)

      $P1 = getprop "tcp", packet_p
      if $P1 goto filter_tcp
      rej_packets_i += 1
      goto filter_end
      filter_tcp:
            $P1 = getprop "tcp_sPort", packet_p
            $P2 = getprop "tcp_dPort", packet_p

            if $P1 == 80 goto http
            if $P2 == 80 goto http
            rej_packets_i += 1
            goto filter_end
            http:
                  acc_packets_i += 1

            filter_end:
             goto loop_read_packet
            loop_read_packet_end:
      ...
```

Výpis 12: Hlavný cyklus filtra v Parrote

6.2.3 ParrotFilter

Na začiatok si zhrnieme niekoľko faktov o filtri. Filter je implementovaný v PIR a spúšťaný priamo z Parrotu. Na svoj beh používa vlastné PMC Packet a PcapFileReader načítané dynamicky. Na volanie potrebných C funkcií využíva NCI, ktoré sú implementované vo vlastnej zdieľanej knižnici *libMapFile2Memory.so*. Názov súboru sa predáva ako parameter príkazového riadku. Samotný filter sa skladá z troch modulov:

1. ParrotFilter.pir – mapuje súbor do operačnej pamäte, číta pakety zo súboru, posiela ich dekodéru a aplikuje filter
2. Decoder.pir – dekóduje pakety, výpis 11 zobrazuje dekódovanie protokolu TCP
3. Library.pir – obsahuje pomocné funkcie

Program sa spúšťa modulom *ParrotFilter.pir/.pbc*, ktorý obsahuje hlavný cyklus. Výpis 12 zobrazuje ukážku zdrojového kódu resp. hlavného cyklu, ktorý prečíta paket zo súboru, pošle ho dekodéru, ktorý nastaví príslušné globálne premenné a nakoniec aplikuje filtračné pravidlo. Jednotlivé moduly môžeme preložiť do PBC (Parrot by ich aj tak v prvom kroku previedol do PBC). Ukážka preloženia a použitia filtra:

parrot Modul.pir -o Modul.pbc
parrot Modul.pbc "cesta k súboru"

7 Virtuálny stroj LLVM

LLVM (10) je, ako samotný názov napovedá, nízko úrovňový virtuálny stroj vyvíjaný pod licenciou Illinois Open Source, ktorá je skoro identická s licenciou BSD. Poskytuje výkonný, statický back-end kompilátor a optimalizáciu programu počas celého jeho života, umožňuje vytvoriť JIT kompilátory, vysoko úrovňové virtuálne stroje, atď. Autorom projektu je Chris Arthur Lattner a Vikram Adve z univerzity Illinois. V roku 2005 Apple najalo Lattnera a sformovalo tým aby pracoval na LLVM pre rôzne účely v rámci ich vývojového systému. V súčasnosti je okrem tohto tímu vyvíjaný a využívaný celou skupinou ľudí z rôznych krajín sveta. Apple využíva LLVM napríklad v OpenGL, OpenCL, pri preklade niektorých výkonnostne kritických častí operačného systému, atď. Okrem Apple využívajú LLVM na rôzne účely aj firmy Adobe Systems Incorporated, Electronic Arts, NVIDIA, Sun Microsystems Laboratories, Siemens, ...

LLVM je bežne dostupný v operačných systémoch typu Linux. Skladá sa z dvoch častí:

1. LLVM súprava – obsahuje nástroje, knižnice, hlavičkové súbory potrebné na používanie nízko úrovňového VM. Obsahuje assembler, disassembler, bitkód analyzátor a optimalizátor, testovacie nástroje.
2. LLVM-GCC (front-end) – poskytuje verziu GCC, ktorá kompiluje C a C++ do LLVM bitkódu.

Pre účely tejto diplomovej práce budeme potrebovať len prvú časť. Na jej stiahnutie resp. inštaláciu môžeme použiť inštalačný manažér danej distribúcie. Opäť sme zvolili postup stiahnutia najnovšej verzie LLVM z domovskej stránky projektu a manuálneho preloženia a inštalácie. Podrobný návod je popísaný na stránkach projektu.

7.1 Architektúra a vlastnosti

LLVM je registrový virtuálny stroj, implementovaný v jazyku C++. Je dostupný pre veľké množstvo platforiem, medzi ktorými samozrejme nechýba Linux s podporou JIT kompilácie (x86 aj x86-64). JIT systém patrí medzi jednu z veľa predností LLVM. Veľa projektov využíva pre svoju potrebu z LLVM práve JIT kompiláciu z dôvodu schopnosti veľmi dobrej optimalizácie programu za behu. Medzi ďalšie vlastnosti LLVM patrí:

1. Kompilačná stratégia navrhnutá na efektívnu optimalizáciu programu počas celého jeho života. LLVM podporuje optimalizáciu pri kompilácii, pri linkovaní, počas behu a dokonca aj offline (po nainštalovaní), zatiaľ čo ostáva transparentná vývojárom a kompatibilná s existujúcimi výstavbovými skriptami.
2. Virtuálna množina inštrukcií je nízko úrovňová reprezentácia strojového kódu, ktorá používa inštrukcie podobné RISC inštrukciám. Poskytuje informácie o toku dát (SSA) a bohaté, jazykovo nezávislé, typové informácie o operandoch. Táto kombinácia umožňu-

je sofistikované transformácie na strojovom kóde, zatiaľ čo ostáva natoľko „malá“, aby sa dala pripojiť k spustiteľným súborom.

LLVM nie je teda v pravom zmysle slova virtuálny stroj (ten je však jeho súčasťou) ale kompilačná infraštruktúra zložená z kolekcie zdrojových kódov, ktoré implementujú jazyk a kompilačnú stratégiu. Primárnymi komponentmi infraštruktúry sú C a C++ frontend založený na GCC, optimalizačný framework pre fázu linkovania obsahujúci rastúcu množinu analýz a transformácií, statický backend pre veľké množstvo architektúr, backend ktorý generuje prenositeľný C kód a JIT kompilátor pre niekoľko architektúr. V rámci tejto diplomovej práce však budeme o LLVM uvažovať predovšetkým ako o virtuálnom stroji.

LLVM síce podporuje napríklad pointre a pointerovú aritmetiku[17], vlákna, ... avšak priamo nepodporuje niektoré vlastnosti ako GC, ktoré by sme normálne očakávali od virtuálneho stroja. Na druhú stranu ponúka k dispozícii prostriedky, ktorými si môžeme chýbajúce vlastnosti doplniť. LLVM teda nechápeme ako alternatívu k virtuálnym strojom typu JVM, CLI a Parrot ale ako doplnok. Od spomínaných VM sa LLVM odlišuje troma kľúčovými vlastnosťami:

1. Nemá poňatie o konštrukciách z vyšších jazykov resp. virtuálnych strojov ako triedy, dedičnosť, výnimky. Obsahuje však mechanizmy na ich realizáciu.
2. Nešpecifikuje behový systém alebo konkrétny objektový model, je nízkoúrovňový natoľko aby behový systém pre určitý jazyk mohol byť implementovaný priamo v LLVM.
3. Negarantuje typovú bezpečnosť, pamäťovú bezpečnosť, jazykovú medzioperabilitu o nič viac ako assembler pre fyzický procesor.

7.1.1 Reprezentácia programu

Reprezentácia programu (11) je jeden z kľúčových faktorov, ktorý odlišuje LLVM od ostatných systémov. Je navrhnutá tak aby poskytla vysoko úrovňové informácie o programoch, ktoré sú potrebné pre sofistikované analýzy a transformácie, zatiaľ čo ostáva natoľko nízko úrovňová aby sme mohli reprezentovať ľubovoľný program.

LLVM aplikácie sa skladajú z modulov. Každý modul pozostáva z funkcií, globálnych premenných, atď. Moduly môžu byť kombinované dohromady za pomoci LLVM linkera, ktorý zlučuje definície funkcií a globálnych premenných, záznamy v tabuľke symbolov a rieši dopredné deklarácie. Každá LLVM aplikácia má tri ekvivalentné formy: binárnu (tzv. bitkód[18], súbory s príponou .bc), v operačnej pamäti a textovú (assembler, súbory s príponou .ll alebo .s).

LLVM používa SSA[19] formu ako primárnu reprezentáciu kódu (výnimku tvoria pamäťové miesta prístupné cez pointre), v ktorej je každý virtuálny register zapísaný práve jednou inštrukciou

[17] Pomocou inštrukcie *getelementptr* viď podkapitolu 7.1.4.
[18] LLVM používa vlastný názov pre bytekód.
[19] http://en.wikipedia.org/wiki/Static_single_assignment_form

a každému použitiu registra dominuje jeho definícia. Predstavme si jednoduchý príklad súčtu dvoch čísel napríklad v jazyku C (viď výpis 13). Použijeme dve premenné, ktoré inicializujeme na 1 a sčítame ich, pričom výsledok priradíme do jednej z nich. Pre jednoduchosť predpokladajme, že v LLVM môžeme tiež inicializovať registre konštantnými hodnotami[20]. Ako si môžeme všimnúť, jazyk LLVM je silno typový, pri každej operácii musíme vždy uviesť dátový typ použitého registra, parametru, ... (v ukážke znamená i32 dátový typ 32 bitové celé číslo). SSA forma sa prejavila v podobe nemožnosti uložiť výsledok súčtu dvoch registrov do jedného z nich, lebo každý register môže byť zapísaný práve jednou inštrukciou (z tejto vlastnosti vyplýva aj neschopnosť uložiť konštantnú hodnotu do registra lebo sa nejedná o inštrukciu). Namiesto toho musíme použiť nový register, ktorý automaticky získa dátový typ použitých operandov. Hlavná výhoda tejto formy je zjednodušenie a zlepšenie rôznych optimalizácií.

```
//Jazyk C
i = 1;
j = 1;
j = i + j;
```

```
;Jazyk LLVM
%i = i32 1
%j = i32 1
;%j = add i32 %i, %j ;nie je možné
%k = add i32 %i, %j
```

Výpis 13: Príklad SSA formy

7.1.2 Typový systém

Unikátna vlastnosť LLVM je jeho typový systém, nemálo podobný systémom, ktoré poznáme z vyšších programovacích jazykov. Práve typovosť umožňuje vykonať množstvo optimalizácii priamo na strednej reprezentácii (IR) bez potreby robiť ďalšie extra analýzy pred transformáciou. Každá inštrukcia, ako už bolo načrtnuté v podkapitole 7.1.1, musí obsahovať informáciu o type použitých operandov, prípadne výsledku. Všetky typové konverzie musia byť explicitne ošetrené na to určenou inštrukciou. Tabuľka 1 poskytuje prehľad o LLVM typovom systéme.

Kategória	Typy
Celé čísla	i1, i2, i3, ... i8, ... i16, ... i32, ... i64, ...
Desatinné čísla	float, double, x86_fp80, fp128, ppc_fp128
Prvá trieda	celé čísla, desatinné čísla, pointer, vektor, štruktúra, union, pole, návestie, metadáta
Primitívne	návestie, void, celé čísla, desatinné čísla, metadáta
Odvodené	pole, funkcia, pointer, štruktúra, zabalená štruktúra (packed structure), union, vektor, opaque

Tabuľka 1: LLVM typový systém

Dátové typy kategórie prvá trieda sú pravdepodobne najdôležitejšie. Inštrukcie môžu produkovať hodnoty iba týchto typov. Primitívne typy tvoria základné stavebné bloky LLVM systému.

[20] V skutočnosti to však nejde, môžeme však použiť trik s inštrukciou súčtu (*%i = add i32 1, 0*), alebo použiť inštrukciu *phi* prípadne alokovať miesto na zásobníku/halde pomocou *alloca*/*malloc* a vložiť hodnotu pomocou inštrukcie *store*, register by však obsahoval pointer a nie danú hodnotu.

Skutočná sila LLVM však pochádza z odvodených typov. Tie umožňujú programátorom reprezentovať polia, funkcie, pointre, atď. Každý z nich obsahuje jeden alebo viac typov prvkov, ktoré môžu byť primitívne alebo odvodené.

7.1.3 Volanie externých funkcií

LLVM v súčasnosti obsahuje dva mechanizmy volania externých funkcií. Prvá možnosť je implementovať tzv. lle_* obaľovaciu funkciu. Tento spôsob je použitý pre niektoré známe a často používané funkcie z knižníc (napr.: *printf*). Druhá možnosť, ktorá je zároveň oveľa pohodlnejšia a jednoduchšia, spočíva v použití JIT subsystému. Ten používa dynamické rozhranie (libffi[21]) na načítanie a volanie funkcií. Do zdrojového kódu musíme uviesť prototypy požadovaných funkcií a potom ich môžeme normálne[22] používať. Je to užitočné predovšetkým pre volanie systémových funkcií, avšak existujú niektoré funkcie, ktoré musia byť ošetrené špeciálne pomocou prvého spôsobu. V prípade, že námi používaná platforma nemá podporu JIT kompilácie, nezostáva nič iné ako použiť prvý mechanizmus.

7.1.4 LLVM inštrukčná sada

Inštrukčná sada LLVM zachytáva kľúčové operácie bežných procesorov ale vyhýba sa strojovo špecifickým obmedzeniam ako fyzické registre, rúry a nízko úrovňové konvencie volania. Poskytuje nekonečnú množinu typových virtuálnych registrov, ktoré môžu udržiavať hodnoty primitívnych typov. LLVM je založené na architektúre načítaj/ulož, program prenáša hodnoty medzi registrami a pamäťou výhradne pomocou operácii load a store za použitia typových pointerov.

Táto časť poskytuje stručný prehľad inštrukcií použitých pri implementácii filtra. Jazyk LLVM ďalej obsahuje veľké množstvo linkovacích typov (napr.: private, common, weak), atribútov funkcií (napr.: nounwind, noreturn), ..., ktoré však už popisovať nebudeme.

Identifikátory a komentáre

Komentáre začínajú symbolom bodkočiarky (;) a pokračujú až do konca riadku. V LLVM rozoznávame dva základné typy identifikátorov: globálne a lokálne. Globálne identifikátory (funkcie, globálne premenné) začínajú znakom @. Lokálne identifikátory (registre a pomenované typy) začínajú znakom %. Existujú tri rôzne formáty identifikátorov:

1. Pomenované premenné - sú reprezentované reťazcom znakov začínajúc daným prefixom *(*napr.: *%foo, @DivisionByZero, %a.really.long.identifier*). Je použitý nasledujúci regulárny výraz: „[%@][a-zA-Z$._][a-zA-Z$._0-9]*“.
2. Nepomenované premenné - sú reprezentované kladnými celými číslami s daným prefixom (napr.: *%12, @2, %44.*).
3. Konštanty – LLVM rozoznáva niekoľko rôznych typov konštánt.

[21] http://en.wikipedia.org/wiki/Libffi
[22] Ako funkcie LLVM, ktoré majú definície priamo v našom zdrojovom kóde.

```
@DEBUG = common global i8 0                          ; globálna premenná
@str = private constant [18 x i8] c"Can`t open file: \00"   ; globálna konštanta – pole znakov
```

Funkcie

Definícia funkcie pozostáva z kľúčového slova *define*, nepovinného typu linkovania, nepovinného typu viditeľnosti, nepovinnej konvencie volania, z návratového typu, nepovinného atribútu parametrov pre návratový typ, z názvu funkcie, zo zoznamu parametrov (každý parameter s nepovinným atribútom), nepovinných atribútov funkcie, nepovinnej sekcie, nepovinného zarovnania, nepovinného názvu GC, zloženej otváracej zátvorky, zoznamu základných blokov a ukončovacej zloženej zátvorky.

```
define [linkage] [visibility]
    [cconv] [ret attrs]
    <ResultType> @<FunctionName> ([argument list])
    [fn Attrs] [section "name"] [align N]
    [gc] { ... }
```

Deklarácia funkcie sa moc nelíši od definície, začína sa kľúčovým slovom *declare* a okrem nepovinných atribútov funkcie, nepovinnej sekcie a tela funkcie začínajúc otváracou zátvorkou je totožná.

Telo funkcie pozostáva zo zoznamu základných blokov, ktoré formujú graf toku riadenia pre funkciu. Každý základný blok môže nepovinne začať návestím, ďalej pokračuje zoznamom inštrukcií a ukončovacou inštrukciou (napríklad inštrukcia skoku alebo návratu z funkcie). Prvý blok funkcie má špeciálne postavenie, je automatický vykonaný po vstupe do funkcie a nemôže mať žiadneho predchodcu (ináč povedané žiadna inštrukcia skoku nemôže na neho odkazovať). Funkcie voláme pomocou inštrukcie *call*, ktorá spôsobí prenesenie riadiaceho toku na špecifikovanú funkciu. Prvá po spustení sa vykonáva, ako sme zvyknutý z iných programovacích jazykov, funkcia *main*.

```
define internal fastcc void @resetOffset() nounwind {   ; definícia funkcie
   entry:                                                ; prvý blok funkcie
      store i32 24, i32* @file_offset
      ret void                                           ; ukončovacia inštrukcia
}
call fastcc void @resetOffset() nounwind                 ; volanie funkcie
```

Terminálne inštrukcie

Ako sme sa zmienili vyššie, každý blok musí byť ukončený terminálnou inštrukciou, ktorá určuje pokračovateľa aktuálneho bloku po jeho ukončení. Existuje sedem rôznych terminálnych inštrukcií, my sa však v rámci DP uspokojíme s dvoma.

- *ret* – slúži na vrátenie toku riadenia a nepovinnej hodnoty z funkcie späť volajúcemu.
- *br* – slúži na prenos riadiaceho toku do iného bloku v rámci danej funkcie. Na jednotlivé bloky sa odkazujeme pomocou kľúčového slova *label* a názvu bloku v tvare lokálnej premennej. Existujú dve základné formy inštrukcie: podmienený a nepodmienený skok.

```
Test:                                          ; názov bloku / návestie
   %cond = icmp eq i32 %a, %b                  ; inštrukcia podmienky (viď nižšie)
   br i1 %cond, label %IfEqual, label %IfUnequal  ; podmienený skok
IfEqual:                                       ; názov bloku / návestie
   ret i32 1                                   ; návrat z funkcie
IfUnequal:                                     ; názov bloku / návestie
   ret i32 0
```

Binárne operácie

Binárne operátory sa používajú na realizáciu väčšiny výpočtov v programe. Vyžadujú dva operandy rovnakého typu nad ktorými vykonajú požadovanú operáciu a vyprodukujú jeden výsledok, ktorý má typ použitých operandov. LLVM pozná nasledujúce binárne operátory: *add, fadd, sub, fsub, mul, fmul, udiv, sdiv, fdiv, urem, srem, frem.* Najviac používaný je operátor *add,* ktorý vráti súčet dvoch operandov.

```
%vys = add i32 4, %var   ; k premennej var pripočíta číslo 4 a výsledok  uloží do %vys
```

Bitové binárne operácie

Bitové binárne operátory slúžia na bitové operácie v programe. Podobne ako normálne binárne operácie vyžadujú dva operandy rovnakého typu a vrátia jeden výsledok typu použitých operandov. Medzi tieto inštrukcie patria:

- *shl* – inštrukcia bitového posunu doľava. Vráti prvý operand posunutý doľava o špecifikovaný počet bitov.

  <result> = shl <ty> <op1>, <op2>

- *lshr* – inštrukcia bitového posunu doprava. Vráti prvý operand posunutý doprava o špecifikovaný počet bitov, pričom sprava dopĺňa nuly.

  <result> = lshr <ty> <op1>, <op2>

- *and, or* – inštrukcie bitového súčinu resp. súčtu. Realizujú bitový logický súčin resp. súčet.

  <result> = and/or <ty> <op1>, <op2>

Operácie adresovania a prístupu do pamäte
Táto kategória zahŕňa inštrukcie na čítanie, ukladanie a alokovanie pamäte. LLVM obsahuje štyri inštrukcie na manipuláciu s pamäťou:

- *alloca* – alokuje miesto v pamäti na zásobníku práve vykonávanej funkcie a automaticky ho uvoľňuje po skončení funkcie. V prípade úspechu je vrátený pointer na alokované miesto.

  <result> = alloca <type>[, i32 <NumElements>][, align <alignment>]

- *load* – načíta miesto odkazované pointrom z pamäte.

  <result> = load <ty> <pointer>[, align <alignment>][, !nontemporal !<index>]*

- *store* – zapíše danú hodnotu na miesto v pamäti odkazované pomocou pointra.

  store <ty> <value>, <ty> <pointer>[, align <alignment>][, !nontemporal !]*

- *getelementptr* – umožňuje pointerovú aritmetiku resp. získať adresu elementov v agregovanej dátovej štruktúre. Vykonáva iba výpočet adresy, do pamäte nepristupuje.

  <result> = getelementptr <pty> <ptrval>{, <ty> <idx>}**

*%0 = getelementptr i8** %argv, i32 1 ; načíta adresu elementu s indexom 1 z poľa argv do %0*
*%1 = load i8** %0 ; načíta hodnotu z pamäte na ktorú odkazuje %0 do %1*

Operácie konverzie
Táto kategória zahŕňa inštrukcie na konverziu typov. Syntax je pre všetky inštrukcie rovnaká, každá vyžaduje jeden operand a nový typ. LLVM pozná nasledujúce inštrukcie konverzie: *trunc, zext, sext, fptrunc, fpext, fptoui, fptosi, uitofp, sitofp, ptrtoint, inttoptr, bitcast.* V rámci implementácie filtra sme použili inštrukciu *zext,* ktorá rozširuje operand nulami. Nový typ musí byť väčší ako stávajúci, pričom bity vyššieho rádu sú zapĺňané nulami.

<result> = instruction <ty> <value> to <ty2>

Ostatné operácie
Do tejto kategórie patria operácie, ktoré sa nedali presne zaradiť. Nás zaujíma predovšetkým inštrukcia podmienky a tzv. phi inštrukcia.

- *icmp* – vracia výsledok porovnania dvoch celých čísel, celočíselných vektorov alebo pointerov. Okrem porovnávaných operandov musíme špecifikovať aj podmienku: *eq* (rovná sa), *ne* (nerovná sa), *ugt* (neznamienkový väčší ako), *uge, ult, ule, sgt* (znamienkový väčší ako), *sge, slt, sle.*

<result> = icmp <cond> <ty> <op1>, <op2>

- *phi* – reprezentuje funkciu φ (Phi) v SSA. Prijíma argumenty vo forme dvojíc: hodnota a názov bloku. Úlohou tejto inštrukcie je rozoznať z ktorého bloku bol predaný riadiaci tok a použiť špecifikovanú hodnotu z danej dvojice. P*hi* inštrukcie môžu figurovať iba na začiatku bloku.

  <result> = phi <ty> [<val0>, <label0>], ...

```
get_packet:
    %acc_packets = phi i32 [ 0, %entry ], [ %8, %acc ], [ %acc_packets, %rej ]
; ak bol predchodcom bloku get_packet blok  entry, nastaví hodnotu %acc_packets na 0
; ak bol predchodcom bloku get_packet blok  acc, nastaví hodnotu %acc_packets na %8
; ak bol predchodcom bloku get_packet blok  rej, ponechá %acc_packets aktuálnu hodnotu
```

7.2 LLVM nástroje

V rámci implementácie filtra a kompilátora sme použili rôzne nástroje LLVM, preto si ich všetky stručne popíšeme na jednom mieste.

- llvm-as (LLVM assembler) – prečíta súbor obsahujúci ľudsky čitateľný assembler a preloží ho do bitkódu. Výsledok zapíše do súboru alebo na štandardný výstup. Pomocou voľby *–o* môžeme špecifikovať výstupný súbor.

  llvm-as assembler.s –o bitkod.bc

- llvm-dis (LLVM disassembler) – konvertuje súbor v binárnej forme na ľudsky čitateľný assembler. Pomocou voľby *–o* môžeme špecifikovať výstupný súbor.

  llvm-dis bitkod.bc –o assembler.s

- llvm-link (LLVM linker) – umožňuje spojiť niekoľko súborov v binárnej forme do jedného bitkód súboru. Najskôr sa pokúša načítať súbory z aktuálnej zložky, ak hľadanie zlyhá pokračuje v zložkách špecifikovaných za voľbou *–L*. Pomocou voľby *–o* môžeme špecifikovať výstupný súbor, pridaním voľby *–S* dostaneme výstup v assemblery namiesto binárnej formy.

  llvm-link modul1.bc modul2.bc modul3.bc –o spojeneModuly.bc

- lli (LLVM interpreter) – priamo vykonáva programy v bitkód formáte. Na podporovaných platformách automaticky použije JIT kompiláciu v opačnom prípade použije interpreter. Pomocou voľby *-force-interpreter={false,true}* môžeme na platformách podporujúcich JIT kompiláciu vynútiť použitie interpretera.

lli spojeneModuly.bc

```
entry:
  %packet = alloca i8* ; vznikne i8**
  %file_address = alloca i8*
  ; namapuje subor do pamate
  %0 = call fastcc i32 @mapFile2Memory(i8** %file_address, i8* %file_path,
       i8 %map_type) nounwind
  %1 = load i8** %file_address
  %2 = icmp sle i32 %0, 0
  br i1 %2, label %error_map, label %get_packet

error_map:
  ; subor sa nepodarilo namapovat
  call void @exit(i32 -1) noreturn nounwind
  unreachable

get_packet:
  %acc_packets = phi i32 [ 0, %entry ], [ %8, %acc ], [ %acc_packets, %rej ]
  %rej_packets = phi i32 [ 0, %entry ], [ %9, %rej ], [ %rej_packets, %acc ]
  ; precita paket zo suboru
  %3 = call fastcc i32 @getPacket(i8** %packet, i8* %1, i32 %0) nounwind
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %end, label %process_packet

process_packet:
  %5 = load i8** %packet
  ; dekoduje paket
  call void @decode(i8* %5, i32 %3) nounwind
  ; aplikuje filtracne pravidlo z modulu filter_rule (mozeme aj vygenerovat)
  %6 = call i1 @filterRule() nounwind ;1 - packet ok 0 - filter packet
  %7 = icmp eq i1 %6, 1
  br i1 %7, label %acc, label %rej

acc:
  %8 = add i32 %acc_packets, 1
  br label %get_packet

rej:
  %9 = add i32 %rej_packets, 1
  br label %get_packet
```

Výpis 14: Hlavný cyklus filtra v LLVM

7.3 Implementácia filtra

Implementácia filtra je založená na princípoch popísaných v kapitole 1 (použitie mmapu, ...) a prebehla bez väčších problémov. Za istých podmienok však zo sebou prináša jeden podstatný rozdiel oproti riešeniam v ostatných jazykoch. Filter realizovaný na platforme x86-64 nie je vždy prenositeľný na platformu x86 (nebavíme sa teraz o binárnej forme). Je to z dôvodu rozdielnych prototypov niektorých systémových funkcií ako napríklad *lseek*, ktorá má na 64 bitových systémoch inú návratovú hodnotu kvôli podpore veľkých súborov (väčších ako 4GB). My sme sa však v rámci diplomovej práce na sporných miestach obmedzili na použitie 32 bitových funkcií. Z toho vyplýva limitované použitie testovacích súborov (menších ako 4GB), ale na

druhú stranu umožňuje beh na obidvoch spomínaných platformách bez úprav kódu. Všetky zdrojové súbory sú na priloženom CD v adresári *LLVM/src/LLVMFilter*.

Samotný filter sa skladá zo štyroch modulov: *run, filter, filter_rule, decoder* a bol navrhnutý tak, aby sme na generovanie filtrovacích pravidiel mohli použiť vlastný kompilátor. Pravidlá sú aplikované pomocou funkcie, ktorá sa nachádza v module *filter_rule (*obsahuje iba túto funkciu). Bližšie informácie o kompilátore a generovaní filtrovacích pravidiel sú v podkapitole 7.4.

```
define internal fastcc void @decode_tcp(i8* nocapture %packet, i32
      %offset) nounwind {
  entry:
    ; nacita adresu z paketu na pozicii offset
    %0 = getelementptr inbounds i8* %packet, i32 %offset
    %1 = load i8* %0
    ; znamienkovo rozsiri hodnotu z typu i8 na typ i16
    %2 = sext i8 %1 to i16
    %3 = shl i16 %2, 8

    %4 = add i32 %offset, 1
    %5 = getelementptr inbounds i8* %packet, i32 %4
    %6 = load i8* %5
    %7 = sext i8 %6 to i16
    %8 = or i16 %3, %7
    ; do globalnej premennej ulozi hodnotu zdrojoveho TCP portu
    store i16 %8, i16* @tcp_src_p

    %9 = add i32 %offset, 2
    %10 = getelementptr inbounds i8* %packet, i32 %9
    %11 = load i8* %10
    %12 = sext i8 %11 to i16
    %13 = shl i16 %12, 8

    %14 = add i32 %offset, 3
    %15 = getelementptr inbounds i8* %packet, i32 %14
    %16 = load i8* %15
    %17 = sext i8 %16 to i16
    %18 = or i16 %13, %17
    ; do globalnej premennej ulozi hodnotu cieloveho TCP portu
    store i16 %18, i16* @tcp_dst_p
    ret void
}
```

Výpis 15: Dekódovanie protokolu TCP v LLVM

Pred tým ako si popíšeme jednotlivé moduly si zhrnieme základné fakty. Filter je kompletne implementovaný v assemblery LLVM a spúšťaný priamo pomocou interpretera *lli* s podporou JIT kompilácie na danej platforme. Jednoduché volanie externých (systémových) funkcií zabezpečuje spomínaný JIT subsystém. Pred spustením programu musíme najskôr jednotlivé moduly preložiť do binárnej formy a potom spojiť do jedného súboru pomocou nástrojov popísaných

v časti 7.2. Za týmto účelom môžeme použiť aj priložený makefile. Cesta k testovaciemu súboru sa zadáva ako parameter príkazového riadku.

Modul run
Úlohou modulu je spracovať argumenty príkazového riadku a spustiť filter (funkciu) s cestou k súboru ako parametrom. Obsahuje funkciu *main,* ktorá sa po spustení vykoná ako prvá.

Modul filter
Úlohou modulu je namapovať súbor do operačnej pamäte, čítať z neho jednotlivé pakety, posielať ich dekodéru a aplikovať filtračné pravidlo. Výpis 14 zobrazuje ukážku hlavného cyklu, ktorý sa volaním príslušných funkcií stará o všetky spomínané akcie.

Modul filter_rule
Tento modul priamo súvisí s vlastným kompilátorom. Obsahuje jedinú funkciu, ktorá pomocou mechanizmu globálnych premenných aplikuje filtračné pravidlo. Celý tento modul je možné vygenerovať kompilátorom na základe zadaného filtra vo vlastnom minijazyku.

Modul decoder
Úlohou modulu je, ako napovedá samotný názov, dekódovanie paketov. Obdrží paket (pointer) od filtra, lokalizuje v ňom príslušné polia a prevedie ich hodnoty z binárnej formy do decimálnej alebo hexadecimálnej podľa potreby. Výpis 15 zobrazuje ukážku dekódovanie protokolu TCP v LLVM.

7.4 Kompilátor minijazyka

V LLVM sme okrem jednoúčelového filtra implementovali aj jednoduchý kompilátor, aby sme mohli posúdiť vhodnosť použitia tohto virtuálneho stroja z každého aspektu. Ako bolo načrtnuté v úvode, v novej generácii modulu NetStat sa počíta s vlastným jazykom na zadávanie filtrov. Dokonca aj v prípade, že by LLVM úplne nevyhovovalo požiadavkám na filtrovanie napríklad z pohľadu výkonu, stále existuje možnosť implementovať jazyk resp. kompilátor založený na tejto technológii. Je to vďaka vlastnostiam popísaným v podkapitole 7.1 ako možnosť vygenerovať binárne súbory z LLVM IR alebo prenositeľný C kód.

Pred samotnou implementáciou sme najskôr navrhli jednoduchý jazyk na zadávanie filtrov tzv. minijazyk. Ten je z časti inšpirovaný jazykom BPF a umožňuje zadať obmedzenia na hodnotu zdrojového a cieľového portu protokolu TCP. Syntax minijazyka je nasledujúca (pozor oddeľovač gramatických pravidiel je znak /, lebo znak | je terminálny symbol):

Start -> LogickýVýraz
LogickýVýraz -> Podmienka / Podmienka & Podmienka / Podmienka | Podmienka

Podmienka[23] *-> Identifikátor = Číslo*
Identifikátor -> TcpSrcPort / TcpDstPort
Číslo -> [0-9]+

```
; rucne napisany filter
@tcp_src_p = linkonce global i16 0
@tcp_dst_p = linkonce global i16 0

; tcp_src_p = 80 or tcp_dst_p = 80
define i1 @filterRule() nounwind {
  entry:
    %0 = load i16* @tcp_src_p
    %1 = icmp eq i16 %0, 80
    br i1 %1, label %http,
      label %next_con

  next_con:
    %2 = load i16* @tcp_dst_p
    %3 = icmp eq i16 %2, 80
    br i1 %3, label %http,
      label %no_http

  http:
    ret i1 1;

  no_http:
    ret i1 0;

}
```

```
; vygenerovany filter
@tcp_src_p = linkonce global i16 0
@tcp_dst_p = linkonce global i16 0

; tcp_src_p = 80 or tcp_dst_p = 80
define i1 @filterRule() nounwind {
  entry:
    %0 = load i16* @tcp_src_p
    %1 = icmp eq i16 %0, 80
    br i1 %1, label %acc,
      label %next_con

  acc:
    ret i1 true

  rej:
    ret i1 false

  next_con:
    %2 = load i16* @tcp_dst_p
    %3 = icmp eq i16 %2, 80
    br i1 %3, label %acc, label %rej
}
```

Výpis 16: Porovnanie filtrov

Úlohou kompilátora je preložiť filter zadaný v minijazyku do assembleru LLVM. Konkrétne je vygenerovaný celý modul (v našom prípade pomenovaný *filter_rule*) s deklaráciami globálnych premenných a definíciou funkcie, ktorá pomocou týchto premenných aplikuje filtračné pravidlo. Pre porty väčšie ako 32767 zadané vo filtri generátor vygeneruje záporné čísla, vnútorne ich však LLVM porovná správne, bit po bite (viď reprezentácia záporných čísel). Výpis 16 porovnáva ručne napísaný kód (naľavo) s vygenerovaným pomocou kompilátora (napravo). Ako si môžeme všimnúť, kódy sa od seba líšia iba pomenovaním blokov. Kompilátor číta konkrétne filtračné pravidlo zo štandardného vstupu, čiže je možné ho zadať aj interaktívne. Koniec zadávania v interaktívnom režime musíme potvrdiť dvojitým enterom. Výsledný kód sa generuje na chybový[24] výstup. Ukážka použitia:

./compiler < input – skompiluje filtračné pravidlo uložené v súbore input napr.: tcpSrcPort = 80 | tcpDstPort = 80
./compiler < input 2> filter_rule.s – skompiluje filtračného pravidlo uložené v súbore input do súboru filter_rule.s

[23] Pri logickom výraze a pri podmienke je potrebné dodržať medzeru medzi neterminálnymi a terminálnymi (&, |, =) symbolmi.
[24] Jedná sa o štandardné chovanie LLVM API.

7.4.1 Implementácia kompilátora

Kompilátor je implementovaný v jazyku C++, pričom na žiadnu jeho časť (lexikálny analyzátor, syntaktický analyzátor) sme nepoužili generátor. V rámci textu nebudeme popisovať realizáciu lexikálneho ani syntaktického analyzátora, ale zameriame sa na LLVM API resp. generovanie výsledného kódu. Zdrojový súbor kompilátora je na priloženom CD v adresári *LLVM/src/LLVMCompiler*. Na jeho preloženie použijeme priložený makefile v adresári *./bin*.

Výsledný kód sa generuje na základe zostaveného abstraktného syntaktického stromu (AST), ktorý zachytáva chovanie programu a uľahčuje nám neskoršie fáze kompilátora ako generovanie kódu. Strom modeluje náš jazyk, pričom pre každú konštrukciu jazyka vytvára jeden objekt. Základ stromu tvorí abstraktná trieda *ExprAST* s čisto virtuálnou metódou *Value *Codegen()* na generovanie výsledného kódu. Všetky ostatné uzly stromu dedia z tejto triedy a implementujú virtuálnu metódu. Jedinú výnimku tvorí logický výraz, ktorý túto metódu prekrýva s návratovým typom *Function**. Keďže náš jazyk obsahuje iba jeden výraz najvyššej úrovne (logický výraz), stačí zavolať po zostavení stromu na tento výraz metódu *Codegen()*. Tá potom generuje svoju časť kódu a volá na príslušné podvýrazy opäť túto metódu, atď. Vďaka spoločnej bázovej triede nemusíme rozlišovať typy jednotlivých výrazov a podvýrazov.

```
//vytvori globalnu premennu @tcp_src_p = linkonce global i16 0
GlobalVariable *tcpSport_G = new GlobalVariable(
       *TheModule, Type::getInt16Ty(getGlobalContext()), false,
       GlobalValue::LinkOnceAnyLinkage, ConstantInt::get(getGlobalContext(),
       APInt(16, 0, false)), "tcp_src_p", 0, false, 0
);
//vygeneruje kod pre globalnu premennu na zaklade objektu tcpSport_G
Value *tcp_src_p_g = Builder.CreateConstGEP1_32(tcpSport_G, 0, "tcp_src_p");
```

Výpis 17: Ukážka generovania kódu pre globálnu premennú

Samotné generovanie kódu zabezpečuje trieda *IRBuilder* z LLVM API. Objekt tejto triedy (*Builder)* musíme pri vytváraní zasadiť do kontextu, v našom prípade nám stačí používať pre celý program jeden[25] globálny kontext. Ako už bolo spomínané LLVM programy pozostávajú z modulov. Na reprezentáciu vytváraného modulu slúži trieda *Module*, ktorá uchováva všetky informácie o programe a samotný výsledný kód. Pomocou metódy *dump* vygenerujeme kód na chybový výstup. Práve tieto dve triedy tvoria základ generovania kódu. Obecne obsahuje API pre každú entitu z LLVM triedu, z ktorých spomenieme aspoň tie základné:

- GlobalVariable – reprezentuje globálnu premennú alebo konštantu. Výpis 17 zobrazuje ukážku generovania kódu pre globálnu premennú.
- Type – slúži na definovanie typov premenných, funkcií, ...
- Value – základná trieda všetkých hodnôt[26] počítaných programom.
- Function – reprezentuje funkciu.

[25] Viacero kontextov sa používa pri viacvláknových aplikáciách.
[26] Hodnotami sú v LLVM aj vygenerované inštrukcie.

8 Vyhodnotenie

Na základe nadobudnutých znalostí získaných počas tvorby diplomovej práce a vykonaných testoch posúdime vhodnosť virtuálnych strojov na dekódovanie paketov. Zameriame sa predovšetkým na zložitosť realizácie jednotlivých riešení a výsledky výkonnostných testov.

Stručne si teraz pripomeňme jednotlivé riešenia. Porovnávame medzi sebou natívny filter napísaný v jazyku C, filter napísaný v jazyku Python využívajúci Scapy (vylúčený však z výkonnostných testov), nástroj tcpdump založený na BPF (iba z pohľadu výkonu), nástroj nbeedump (bez JIT kompilácie) realizovaný pomocou knižnice NetBee, filter implementovaný vo virtuálnych strojoch LLVM a Parrot (bez JIT kompilácie). Pri výkonnostných testoch nebude použitý kompilátor z vlastného jazyka do LLVM IR, ktorý by zbytočne skresľoval výsledky.

8.1 Zložitosť realizácie

Posúdenie náročnosti realizácie vychádza zo skúsenosti nadobudnutých počas implementácie jednotlivých riešení. Hodnotiť budeme všetky aspekty tvorby programu od editácie kódu až po dokumentáciu. Všetky riešenia boli programované v integrovanom vývojom prostredí Eclipse s použitím dostupných rozšírení na zvýrazňovanie syntaxe. V priebehu celej diplomovej práce sme poskytovali ukážky rovnakých častí zdrojových kódov realizovaných pomocou rôznych riešení. Na ich základe si môže čitateľ urobiť vlastný obraz o niektorých aspektoch tvorby filtra (syntax, čitateľnosť, vyjadrovacie schopnosti jednotlivých jazykov). Na ohodnotenie si zavedieme jednoduchú stupnicu od 1 do 3, kedy 1 znamená jednoduchá a 3 znamená veľmi náročná realizácia.

Ako prvý sme implementovali filter v jazyku v C. Tento jazyk patrí všeobecne medzi najpoužívanejšie a v rukách dobrého programátora sa jedná o veľmi mocnú zbraň. Existuje o ňom nespočetne mnoho knižných publikácií, webových stránok plných návodov a ukážok. Takmer každý sofistikovanejší textový editor podporuje zvýrazňovanie syntaxe kódu. Aj keď implementácia filtra bola pomerne jednoduchá a priamočiara, vyžaduje si minimálne pokročilé znalosti tohto jazyka (pointre, pointerová aritmetika, bitové operácie). Jednalo sa však o veľmi nenáročnú aplikáciu (jednovláknovú), komplexnejšie aplikácie si vyžadujú profesionálne znalosti. Integrácia programov s operačným systémom typu Linux resp. volanie systémových knižníc je triviálne, keďže samotné knižnice sú napísané v tomto jazyku. Pred spustením aplikácie je potrebné zdrojové kódy preložiť a spojiť, čo značne spomaľuje vývojový cyklus. Realizáciu filtra hodnotíme známkou 2, čiže náročná.

Ďalej nasledovala implementácia filtra v jazyku v Python. Tento jazyk je medzi skriptovacími jazykmi veľmi obľúbený. Existuje o ňom veľké množstvo knižných publikácií, webových stránok s návodmi a ukážkami. Veľa textových editorov podporuje zvýrazňovanie syntaxe tohto jazyka. Samotné naprogramovanie filtra bolo zo všetkých riešení najjednoduchšie, predovšetkým vďaka použitiu projektu Scapy a samotnému faktu, že sa jedná o skriptovací jazyk určený

na rýchle programovanie. IDE Eclipse upozorňuje na chyby v zdrojom kóde už počas písania kódu čo značne urýchľuje vývoj. Výsledný program nie je nutné kompilovať do strojového kódu. Ako už bolo spomínané v kapitole 3, jazyk je však absolútne nevhodný pre programy závislé na výkone. V oblasti spracovania paketov je použiteľný napríklad na jednorazové spracovanie malého množstva dát. Realizáciu filtra hodnotíme ako jednoduchú, čiže známkou 1.

Knižnicu NetBee budeme hodnotiť skôr z pohľadu použitia ako implementácie. Jedná sa o projekt priamo zameraný na analyzovanie a spracovanie sieťovej prevádzky. V rámci diplomovej práce sme použili nástroj *nbeedump* implementovaný v jazyku C++. NetBee poskytuje API na spracovanie paketov, ktoré nás abstrahuje od technológií použitých vo vnútri. Bohužiaľ je však API nepoužiteľné ako samotná dokumentácia. Názvy tried použitých v zdrojov kóde sa často líšia od tried v API, niektoré príklady dokonca nebolo kvôli tomu možné ani preložiť. Keby nebol nástroj *nbeedump* už naimplementovaný, bolo by veľmi prácne a zdĺhavé realizovať filter pomocou tohto riešenia. Použiteľnosť tohto projektu sa tak obmedzuje na inšpiráciu v podobe použitej architektúry a ideí. Na základe týchto dôvodov hodnotíme NetBee známkou 3, teda veľmi náročná realizácia.

Konečne sa dostávame k prvému virtuálnemu stroju, k Parrotu. Implementácia filtra nebola podložená žiadnou predchádzajúcou skúsenosťou s programovaním v assemblery resp. v jazyku PIR. Pred samotnou realizáciou bolo potrebné podrobne preštudovať dokumentáciu k Parrotu, vďaka čomu môžeme poskytnúť veľmi objektívne ohodnotenie. Dokumentácia k projektu je pomerne obsiahla, bohužiaľ jej však chýba lepšie logické usporiadanie, niektoré informácie sme našli na miestach, kde by sme ich nečakali. Návod k programovaciemu jazyku PIR je kvalitne spracovaný a poskytuje všetky základné informácie potrebné pre začiatočníkov. Horšie je do s dokumentáciou k niektorým častiam ako PES, PMC, NCI, ktoré boli často neaktuálne a pomerne neprehľadné. Keby sa autori projektu viac zamerali na dokumentáciu ako na neustále vydávania nových verzií, prispelo by to podľa nás k skvalitneniu projektu. Jazyk PIR je pomerne jednoduchý, predovšetkým vďaka syntaktickým vymoženostiam z vyšších programovacích jazykov, podpore objektových princípov. Je preto vhodný ako na ručné písanie tak aj na generovanie kompilátorom. Ladenie kódu je pomerne náročnejšie, Parrot však obsahuje vlastný debuggovací nástroj. Zvýrazňovanie syntaxe v čase implementácie filtra bolo dostupné pre editory Vim a Emacs, v súčasnosti pribudla aj podpora pre Kate. Keďže PIR nepodporuje priamo pointerovú aritmetiku, bolo potrebné naimplementovať vlastné dátové typy v jazyku C. Zdrojové kódy nie je potrebné kompilovať ani spájať. Ako sme spomínali v kapitole 6, projekt obsahoval niekoľko nedostatkov a na jeho použitie je potrebné ho preložiť ručne s menšími úpravami. Parrot nakoniec po hlbšej úvahe hodnotíme známkou 3, teda veľmi náročná realizácia, aj keď je to veľmi tesné.

Pri virtuálnom stroji LLVM budeme okrem tvorby filtra v LLVM IR hodnotiť aj tvorbu kompilátora v C++ z vlastného minijazyka do LLVM IR. Tak ako v predchádzajúcom príklade sa implementácia filtra v LLVM IR neopierala o žiadnu predchádzajúcu skúsenosť s týmto jazy-

kom. Dokumentácia k LLVM je aktuálna a kvalitne spracovaná, neuškodil by však podrobnejší návod v podobe ukážok a základných konštrukcií k LLVM IR, ktorý je obmedzený len na podrobné vymenovanie inštrukcií. Syntax jazyka je pomerne zložitá a jej naštudovanie zabralo viac času v porovnaní s PIR. Neustále uvádzanie znakov (% lokálne, @ globálne) pre premenné, funkcie (ich atribúty) a špecifikovanie dátových typov znepriehľadňujú zdrojový kód (predovšetkým volanie funkcií je niekedy ťažko čitateľné) a spomalili implementáciu filtra. Autori projektu však na to majú svoje dôvody, jedná sa hlavne o množstvo sofistikovaných analýz a optimalizácii. Ladenie programu je náročné, LLVM obsahuje za týmto účelom vlastný debuggovací nástroj. Zvýrazňovanie syntaxe je dostupné v editoroch Emacs a Vim. Samotný program je pred spustením potrebné preložiť do binárnej formy (bitkódu) a spojiť, čo spomaľuje vývojový proces. Integrácia s operačným systémom typu Linux resp. volanie externých C a C++ funkcií je jednoduché vďaka JIT subsystému. Implementácia kompilátora bola o poznanie jednoduchšia. Autori projektu spracovali veľmi pekný a podrobný návod ako napísať jednoduchý kompilátor. API pre tvorbu kompilátora je prehľadné, názvy tried odpovedajú realite a ich použitie je veľmi intuitívne. Celkovo hodnotíme implementáciu filtra v LLVM IR známkou 3, teda veľmi náročná realizácia a implementáciu kompilátora ako náročná, čiže známkou 2.

8.2 Výkon

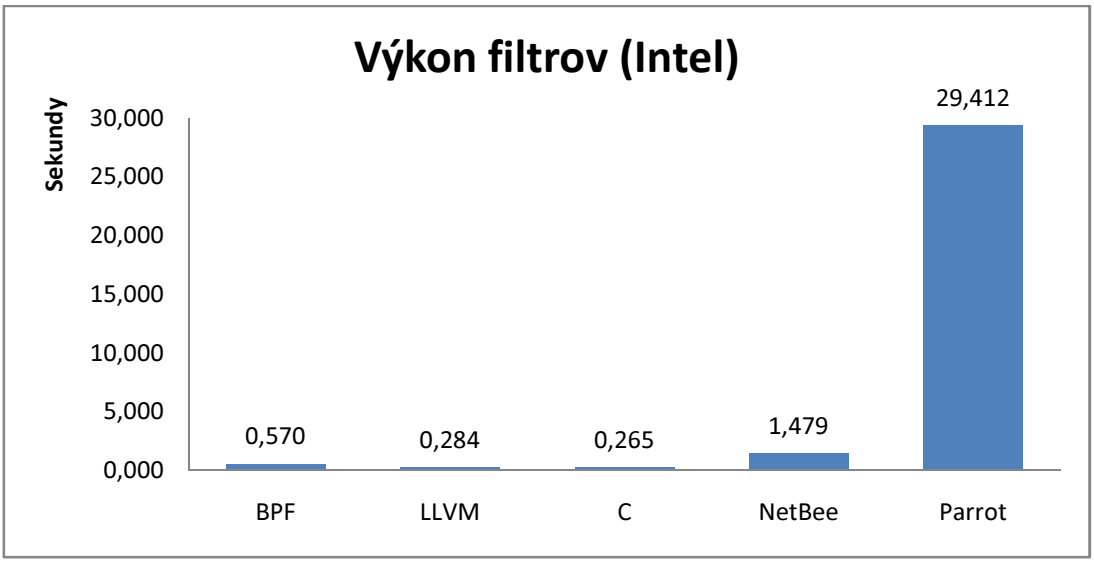
Výkon jednotlivých riešení sme testovali na spracovaní 800 MB súboru vo formáte knižnice PCAP vytvoreného z reálnej sieťovej komunikácie. Filtrovali sme pakety so zdrojovým a cieľovým portom 80 protokolu TCP . Testy boli vykonané na dvoch zostavách:

1. Intel Core2 Duo CPU E8400 @ 3.00 GHz, 4 GB RAM, x86-64 špecifická distribúcia firmy LinuxBox.cz založená na CentOS s jadrom 2.6.34
2. AMD Athlon Dual Core Processor 4850e 1000 MHz, 4 GB RAM, x86-64 distribúcia Fedora 12 s jadrom 2.6.32.11

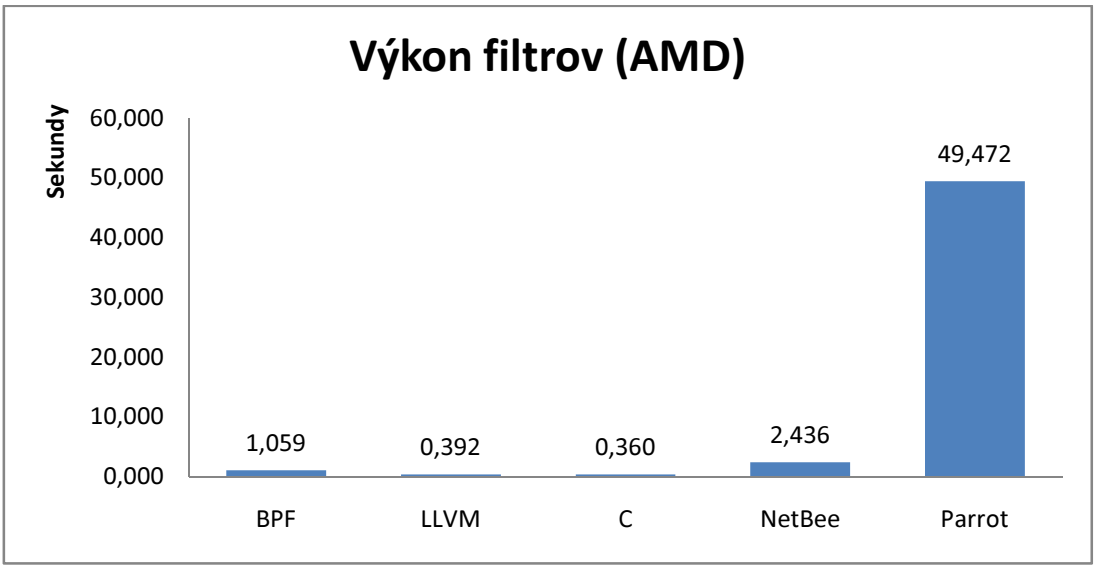
Testy simulujú real-time zachytávanie sieťovej prevádzky, pri ktorej je naplno využívané jedno jadro procesora a 1GB operačnej pamäte. Prístupy k disku sú obmedzené na počiatočné namapovanie celého súboru do operačnej pamäte. Prvý beh každého filtra bol ignorovaný, aby sa súbor natiahol do pamäte a ostal uložený vo vyrovnávacej pamäti. Takto sa réžia *mmapu* (čítanie z disku) znížila na 0 resp. bola nemerateľná a programy využívali vyrovnávaciu pamäť. Každý filter bol spustený 10 krát na každej zostave, výnimku tvorí Parrot, ktorý bol pustený len 5 krát. Čas behu jednotlivých filtrov sme merali pomocou nástroja *time*.

Na obrázkoch 9 a 10 sú výsledky výkonnostných testov, ktoré na obidvoch platformách dopadli rovnako. Najlepší výkon dosiahol očakávane natívny filter v jazyku C. Druhé miesto obsadil prekvapivo filter realizovaný pomocou LLVM, ktorý je na Inteli pomalší približne o 7 % a na AMD o 9 %. Dá sa predpokladať, že výkon filtra realizovaného v LLVM má ešte rezervy. Predsa len sa jednalo o našu prvú skúsenosť s týmto virtuálnym strojom, zároveň hlbšia znalosť a použitie linkovacích typov, vlastností funkcií, ... by mohli priniesť zlepšenie. Na treťom mies-

te sa umiestnil BPF filter (tcpdump). Lepšie výsledky filtra v C aj LLVM pripisujeme ich jednoduchosti, predsa len nástroj tcpdump je veľmi komplexná aplikácia. Prekvapivo dobré výsledky dosiahol aj NetBee filter (nbeedump), napriek tomu, že nebola použitá JIT kompilácia (nepodporuje 64 bitové systémy). Posledné miesto obsadil očakávane Parrot filter z ktorého bol z verzie na verziu úplne odstránený JIT subsystém.



Obrázok 9: Porovnanie výkonu filtrov (Intel)



Obrázok 10: Porovnanie výkonu filtrov (AMD)

9 Záver

Cieľom diplomovej práce bolo posúdiť vhodnosť virtuálnych strojov na dekódovanie paketov. Myslím, že sa mi stanovený cieľ podarilo splniť do posledného bodu. Na základe tejto práce môžeme povedať, že virtuálne stroje obecne sú vhodné na dekódovanie paketov. V podstate to dokazuje samotný projekt NetBee, kde sa autori vydali cestou implementácie virtuálneho stroja simulujúceho sieťový procesor. Ďalším dôkazom je BPF, ktoré používa virtuálny stroj na aplikovanie filtračných pravidiel. Obidve tieto riešenia dosahujú dobrý výkon pri filtrovaní sieťovej prevádzky.

V rámci diplomovej práce sme testovali konkrétne virtuálne stroje: Parrot a LLVM. Keby sme ich porovnali medzi sebou, Parrot jednoznačne ťahá za kratší koniec po každej stránke. Menej kvalitná dokumentácia, chyby v projekte, odstránenie JIT subsystému jasne znižujú kvalitu Parrotu. Dokonca aj projekt NetBee, ktorý bol tiež testovaný bez JIT kompilácie dosiahol niekoľko násobne lepšie výsledky. Bolo by zaujímavé otestovať Parrot s JIT subsystémom, avšak samotní autori projektu nemajú jasnú predstavu o jeho budúcnosti. Záver je teda jasný, Parrot sa v žiadnom prípade nehodí na dekódovanie paketov.

Na druhú stranu nás veľmi milo prekvapil virtuálny stroj LLVM. Kvalitne spracovaná dokumentácia a výkon porovnateľný s jazykom C hovoria samé za seba. Aj keď realizáciu filtra v assemblery LLVM sme ohodnotili ako veľmi náročnú, nejedná sa o veľkú prekážku. V budúcnosti sa predpokladá implementácia sofistikovaného kompilátora a nie programovanie priamo v assemblery tohto virtuálneho stroja. Za týmto účelom sme implementovali jednoduchý kompilátor, ktorý poslúži ako odrazový mostík pre budúcich programátorov. Realizáciu kompilátora sme ohodnotili z pohľadu zložitosti rovnako ako implementáciu filtra v jazyku C. Podľa môjho názoru sa tak viac oplatí investovať úsilie do realizácie kompilátora pre LLVM, ktorý by generoval dekodér a filter ako do implementácie týchto častí v jazyku C.

Záverom diplomovej práce je teda jednoznačné odporúčanie použitia virtuálneho stroja LLVM na dekódovanie a filtrovanie sieťovej prevádzky. Moja predstava ideálneho riešenia v sebe zahŕňa použitie ideí knižnice NetBee ako využitie vlastného jazyka pre popis protokolov a jazyka pre zadávanie filtrov spolu s výkonom LLVM.

Citovaná literatúra

1. Virtual machine. *Wikipedia, the free encyclopedia.* [Online] [Dátum: 12. September 2009.] http://en.wikipedia.org/wiki/Virtual_machine.

2. Libpcap File Format. *Wireshark Wiki.* [Online] [Dátum: 22. September 2009.] http://wiki.wireshark.org/Development/LibpcapFileFormat.

3. **Biondi, Philippe.** *Scapy Documentation.* [PDF] 2009.

4. Berkeley Packet Filter. *Wikipedia, the free encyclopedia.* [Online] [Dátum: 30. September 2009.] http://en.wikipedia.org/wiki/Berkeley_Packet_Filter.

5. **Iliofotou, Marios.** *TCPDump filters.* [PDF] 2009.

6. *The NetBee Library.* [Online] [Dátum: 2. Október 2009.] http://www.nbee.org/doku.php.

7. **Degioanni, Loris, a iní.** *Network Virtual Machine (NetVM): A New Architecture for Efficient and Portable Network Applications.* Záhreb, 2005.

8. **Morandi, Olivier, a iní.** *Enabling Flexible Packet Filtering Through Dynamic Code Generatio.* Peking, 2008.

9. *Parrot VM.* [Online] [Dátum: 15. Október 2009.] http://www.parrot.org/.

10. JITRewrite. *Parrot Developer Wiki.* [Online] [Dátum: 25. Január 2010.] http://trac.parrot.org/parrot/wiki/JITRewrite.

11. *The LLVM Compiler Infrastructure.* [Online] [Dátum: 13. Február 2010.] http://llvm.org/.

12. **Lattner, Chris a Adve, Vikram.** *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* [PDF] Palo Alto, California, 2004.

A. Obsah CD

Ku riešeniam, ktoré si vyžadujú kompiláciu sú priložené aj binárne súbory preložené pod špecifickou distribúciou Linuxu (založená na CentOS), ktorú používa firma LinuxBox.cz. Na preloženie súborov pod inou distribúciou slúžia priložené makefily. K NetBee, LLVM a Parrot je stiahnutá kompletná dokumentácia prezerateľná offline, spakovaná vo formáte ZIP.

/	*- koreňový adresár*
/C/PacketFilter	*- zdrojové súbory filtra v jazyku C*
/LLVM/src/LLVMCompiler	*- zdrojové súbory vlastného kompilátora pre LLVM*
/LLVM/src/LLVMFilter	*- zdrojové súbory filtra v LLVM IR*
/LLVM/src/llvm-2.6.tar.gz	*- zdrojové súbory VM LLVM*
/Netbee/src/netbee.tgz	*- zdrojové súbory knižnice NetBee*
/Netbee/src/netpdl-min.xml	*- min. verzia databáze s protokolmi pre NetBee*
/Netbee/src/readme-compile.htm	*- návod na preloženie NetBee*
/Parrot/src/MapFile2Memory	*- zdrojové súbory zdieľanej knižnica pre Parrot filter*
/Parrot/src/ParrotFilter	*- zdrojové súbory Parrot filtra*
/Parrot/src/PMC	*- zdrojové súbory PMC pre Parrot filter*
/Parrot/src/parrot-1.6.0.tar.gz	*- zdrojové súbory Parrotu verzie 1.6.0*
/Parrot/src/parrot-2.0.0.tar.gz	*- zdrojové súbory Parrotu verzie 2.0.0*
/pcapfile/minitest.pcap	*- malý testovací súbor*
/Python/PythonFilter	*- zdrojové súbory filtra v jazyku Python*
/testy	*- výsledky testov*
/zdroje	*- použité zdroje v DP*
/Diplomová práca.docx	*- text DP vo formáte Word 2007*
/Diplomová práca.pdf	*- text DP vo formáte PDF*